DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

DAMAGE ANALYSIS OF INTERNAL FAULTS IN FLUX CONCENTRATING PERMANENT MAGNET MOTORS

by
Francis R. Colberg

B.S. Elec. Eng., University of Puerto Rico, (1977)

Submitted to the Department of
OCEAN ENGINEERING
and to the Department of
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in Partial Fulfillment of the Requirements for the Degrees of

NAVAL ENGINEER

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1994

©Francis R. Colberg

The author hereby grants to M.I.T. and to the U.S. Government permission to reproduce and to distribute copies of this thesis document in whole or in part.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

DAMAGE ANALYSIS OF INTERNAL FAULTS IN FLUX CONCENTRATING PERMANENT MAGNET MOTORS

by

Francis R. Colberg

Submitted to the Department of Ocean Engineering and the Department of
Electrical Engineering and Computer Science on May 6, 1994 in partial fulfillment
of the requirements for the Degrees of Naval Engineer and Master of Science in
Electrical Engineering and Computer Science

Abstract

It is the purpose of the proposed research to develop a digital computer simulation model to study the effects of an internal fault in a large permanent magnet ac synchronous motor. Permanent magnet motors are being considered as an alternative for ships with electric propulsion systems. In an electric propulsion system a large motor will be directly connected to a propulsion shaft. A windmilling shaft will continue to turn the rotor of the propulsion motor after the motor has been disconnected from its electrical power supply source.

Following an internal electrical fault in a propulsion motor, it is expected that the motor will be disconnected from its electrical supply source. With the ship operating at or near rated speed following a casualty to the propulsion plant, the ship will coast down to a stop or until the crew takes action to stop the ship. A windmilling permanent magnet motor will generate a large enough internal voltage to continue to support large fault currents.

This research will focus on the fault transient and the motor behavior during the time that the propulsion shaft is windmilling. Shorting the motor terminals will be considered as a means of reducing the power input into the fault.

Thesis supervisor: Dr. James L. Kirtley, Jr.
Title: Associate Professor of Electrical Engineering

Acknowledgments

I wish to thank my advisor, Professor James L. Kirtley for his support and guidance, without whom I would not have been able to complete this work. I would also like to thank CAPT Al Brown and LCDR Jeff Reed for their intellectual support and Professor A. Douglas Carmichael, my thesis reader.

I would like also to thank my parents and my wife's parents for their support and encouragement. Finally, I want to thank my wife, Maribeth for her support and assistance throughout my graduate work.

I wish to dedicate this thesis to my children, Barbara and Steffen for understanding and putting up with me during the past three years.

Table of Contents

Abstract 2

Acknowledgments 3

Table of Contents 4

Chapter 1. Introduction 6
- 1.1 Ship Propulsion Systems 8
- 1.2 Permanent Magnet Motors 9
- 1.3 Ship Model 11
- 1.4 Faults in Permanent Magnet Ship Propulsion Motors 12
- 1.5 Research Approach 13

Chapter 2. Conceptual Design 15
- 2.1 Air gap size 19
- 2.2 Magnet dimensions 19
- 2.3 Determination of terminal current and machine rating 21
- 2.4 Machine Reactances 22
- 2.5 Internal Voltage 23
- 2.6 Stator sizing 25
- 2.7 Stator leakage reactance 26
- 2.8 Losses and machine efficiency 27
- 2.9 Back iron sizing 28
- 2.10 Weight Calculations 29

Chapter 3. Dynamic Models 30
- 3.1 Two axis transformation 30
- 3.2 Per Unit Scaling 32
- 3.3 Permanent Magnet Motor Model 33
- 3.4 Network model 39
- 3.5 Fault Model 39
- 3.6 Motor Load Model 41

Chapter 4. Simulation Model 44
- 4.1 Discussion of the simulation program 44
- 4.2 Fault simulation 47
- 4.3 Simulations 50
- 4.4 Fault Parameters 51

Chapter 5. Conclusions and Results **53**
5.1 Simulation Results 53
5.1.1 Simulation with motor terminals open 53
5.1.2 Simulation with shorting of the motor terminals 59
5.3 Suggestions for Future Research 65
5.4 Conclusions 65

References **67**

Appendix A. Motor Design Spreadsheet **69**

Appendix B. Node and Network Pre-Processor **71**

Appendix C. Load Flow Program **76**
Appendix C-1. Load flow calculation program 78
Appendix C-2. Node Voltage Calculation 82
Appendix C-3. Line Admittance Calculation 84

Appendix D. Line Simulation Input File **85**

Appendix E. Synchronous Machine / Network Simulator **92**
Appendix E-1. Motor Object 109
Appendix E-2. Network Program 124

Chapter 1. Introduction

The use of electric ship propulsion offers significant advantages in ship design, construction and operation. Placing electric propulsion motors as far back in the ship as possible serves to reduce long shaft lengths. Propulsion shafts often go through many compartments, creating design complications with shaft line component alignment and compartment arrangement. In addition, electric propulsion can reduce the number and type of prime movers in the ship. Propulsion power and shipboard electrical power can be derived from common prime movers.

In electric propulsion ships, prime movers can be located anywhere in the ship. This flexibility can improve survivability and make maintenance and shipboard arrangements easier [1].

Using electric drive can provide the ship with greater operational flexibility. Electric drive ships can operate the prime movers at their optimal speed and most efficient speed. By operating the prime movers at their most efficient speed, the ship's fuel consumption can be lowered. This could decrease the frequency of refueling and increase the endurance of the ship.

Electric power propulsion has been used in ships, commercial and military, for over 50 years. However, some drawbacks have been higher costs and greater weight than mechanical drive systems. Advanced systems using permanent magnets have the potential of being lighter than similar conventional systems.

The US Navy is developing modern electric drive propulsion systems for its ships. In some of the conceptual designs for these electric propulsion systems, the use of permanent magnet propulsion motors and ship service generators has been considered. An issue that needs to be investigated is the behavior of such machines during and following an electrical fault inside the motor or generator, where the machine supply breakers might not be able to prevent damage. Of concern are internal arcing faults that can result in very large currents and high localized temperatures [2]. These large currents and high temperatures can cause extensive damage to insulation and conductors and more importantly to the permanent magnet themselves.

The arcing process itself can cause significant damage to the machine. Arcs can compromise the electrical, as well as, the mechanical integrity of the machine. Localized damage caused by the electric arc, such as pits or hardened spots, can serve as crack initiators from which cracks that can be induced by machine vibrations can initiate and propagate [3].

After a permanent magnet generator or motor is disconnected from the rest of the system it will continue to generate an internal voltage, E_{af}, until the field comes to a stop. In wound field machines the field circuit breaker can be tripped simultaneously with the main circuit breakers thus essentially eliminating the field. Figure 1 shows a simple schematic of an ac synchronous machine.

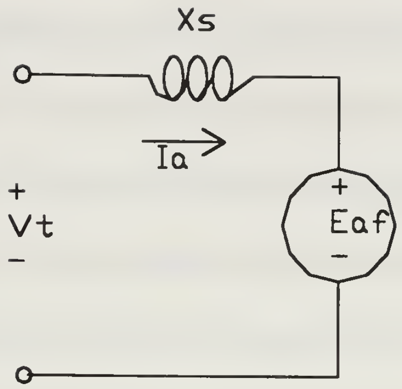

Figure 1-1 Simplified Motor Model

The internal voltage generated in the machine is proportional to the rotational speed of the field. In big machines, such as those used for propulsion or power generation, these internal voltages can be very large. The generated internal voltage can be large enough to continue the arcing process and cause further damage while the machine is coasting down.

The purpose of this research is to develop some tools that can be used to investigate what happens to a large permanent magnet motor following an internal fault while the motor coasts down.

1.1 Ship Propulsion Systems

Current naval propulsion systems consist of multiple diesel engines or gas turbines coupled to one or more propulsion shafts through a set of reduction gears and clutches. Using two engines per shaft provides redundancy and continuity of propulsion. In the case of damage or maintenance to one engine while the ship is underway, propulsion can still be maintained on that shaft.

Although mechanical systems are very simple and reliable, they have some major disadvantages. Mechanical drive systems require separate prime movers for electrical power generation. Alignment between the shaft and the prime mover requires that prime movers be located as low in the ship as possible. This requirement results in large amounts of "lost volume" inside the ship and the superstructure for intake and exhaust trunks. Some of this volume can be recovered in ships with electric drive.

Relatively light prime movers, such as gas turbines, can be located higher in the ship as electric drive does not require alignment between the propulsion motor and the prime mover. Conceivably, prime movers could be located in the superstructure. Locating the prime movers higher in the ship can minimize the arrangeable volume lost to long exhaust and intake trunks. Other advantages and drawbacks of these systems are discussed in [4].

Another possibility that electric propulsion offers is the capability to move the propulsion motors outside the hull of the ship.

1.2 Permanent Magnet Motors

The machine used in this research is a permanent magnet ac synchronous motor. This is essentially an ac synchronous motor in which the field windings have been replaced by permanent magnets. The analysis used for this machine is that used for wound field ac synchronous machines assuming that the machine is excited by a field current of constant value.

Permanent magnet machines have a number of advantages over wound field machines. One of their most significant advantages is the elimination of the field windings

and the power losses generated in these windings. In addition, since no electrical connections are required to supply the field, this eliminates the need for slip rings and brushes. Therefore, eliminating the field windings can also result in smaller machines than wound field machines. This is especially important in marine applications that are volume limited where space is critical.

Some of the limitations of permanent magnet machines come from the permanent magnets themselves. Excessive currents in the motor windings or excessive heat could result in demagnetization of the magnets. Other limitations and advantages of permanent magnet machines are discussed in detail in [5] and [6].

The control aspects and the electrical power requirements of permanent magnet ac machines are not addressed in this research. Such aspects of permanent magnet and other electrical machines are addressed in references such as [7] and [8].

For the purposes of this research the permanent magnet motor will be assumed to have been operating in steady state at the time the fault is initiated. After the fault is initiated the motor will be disconnected from the rest of the network and allowed to coast down. This research will only address large, low speed motors such as those that would be used for ship propulsion. In this configuration the propulsion shaft will be directly connected to the rotor of the machine.

The permanent magnet motor used in this research is a notional machine. A notional machine was used as this type of propulsion machinery is not yet in use in naval ships. Design and sizing calculations of this machine are discussed in chapter two.

One of the advantages of electric drive propulsion is the ability to directly drive a propulsion shaft without the need for reduction gears. It is desirable that ship propulsion motors operate at low speeds, so these machines will have a large number of poles. As the number of poles increases the physical size of the machine will increase. During the notional design phase an effort was made to keep the size and weight of the motor comparable with current propulsion machinery. For this research a flux concentrating permanent magnet machine was selected. The geometry of this machine is shown and discussed in more detail in chapter two.

1.3 Ship Model

Following the fault to the motor, after the main supply breakers to the motor are open, the ship is assumed to coast down to some fraction of its initial speed. During this coast down period and during the initial phases of the casualty it will be assumed that no action is taken by the ship's crew to slow down or stop the shaft. The basis for this assumption is that during the initial phase of the casualty, the first indication available to the operators will be the tripping of the main supply breakers. Once the main supply breakers trip, there will exist an inherent time delay before action is taken to stop the affected shaft and take corrective actions. This delay is due to the time that it will take for ship's personnel to evaluate, recognize and act to combat the casualty.

Automatic protection systems, that operate when the breakers trip, could be used to minimize the time delay in stopping the shaft. A problem with these systems is that they could initiate protective action during spurious breaker trips. During these trips protection is not necessary and could be detrimental to the operation of the ship.

The mechanical energy supplied to the motor while the ship is coasting down is proportional to the speed of the shaft squared. It can be shown that for a given propeller the rotational speed of a windmilling shaft is proportional to the speed of the ship. As the ship coasts down the mechanical energy supplied to the motor will decrease as the ship slows down.

1.4 Faults in Permanent Magnet Ship Propulsion Motors

The focus of this research is investigating the performance of large permanent magnet motors such as those that would be used for ship propulsion. It can be expected that part of the electrical protection of these large motors will be main supply breakers. These breakers will disconnect the motor from its electrical power supply upon detection of a fault in the motor.

Internal faults are of special interest due to the large amounts of energy that can be supplied to these motors by a free spinning shaft after the electrical supply breakers are tripped. Since the field of the machine is supplied by the permanent magnets, the motor with the free spinning shaft will behave like a generator supplying power to the fault. Referring to figure 1-1, the excitation voltage, E_{af}, is proportional to the flux produced by the field, ϕ_f, and the frequency of the field excitation, ω [5]

$$E_{af} = K\omega\phi_f \qquad (1\text{-}1)$$

In permanent magnet machines a constant field flux, ϕ_f, is supplied by the permanent magnets. Therefore to stop this generator action, the shaft must be stopped. Stopping the shaft could be accomplished by using a mechanical shaft brake, changing the

pitch of the propeller in ships with controllable pitch propellers or stopping or slowing down the ship. Ships with more than one propulsion shaft can use the unaffected shaft to slow down or stop the ship.

Other ways to slow down or stop permanent magnet machines are discussed in [9]. These methods are for unfaulted machines whose shafts are not being driven by an external source such as a ship's windmilling propeller. The windmilling propeller and shaft will rotate at a speed that is proportional to the ship's speed.

1.5 Research Approach

This research studied the dynamic behavior of permanent magnet ac synchronous motors following an internal fault. The motor studied was assumed to be directly connected to a ship's propulsion shaft. Following the fault, the motor was disconnected from its electrical supply source and allowed to windmill as the ship coasted down to a fraction of its initial speed.

Different from wound field machines, permanent magnet machines have a constant field flux supplied by the magnets. As the shaft windmills this constant field flux will generate an internal voltage in the machine. A sufficiently large internal voltage can continue to support an internal fault in the machine as the shaft windmills.

To accomplish the goals of this research the following tasks were identified and performed:

1. A notional permanent magnet motor having the performance requirements of a ship's propulsion motor was derived. To do this task, a motor design spreadsheet was

developed. By specifying the principal motor requirements the motor parameters were estimated.

2. A dynamic model of the permanent magnet motor, which incorporated the parameters of the notional design was derived.

3. Models for the internal fault and of the ship were derived.

4. The models of the motor, fault and ship were incorporated into a dynamic simulation program.

5. The simulation was run and the results evaluated.

Chapter 2. Conceptual Design

This chapter describes the procedure used to design the notional permanent magnet motor used in this research. The motor designed incorporates desired attributes of a motor for naval propulsion. The procedure described in this chapter is implemented into an Excel [10] spreadsheet, Appendix A. For the motor design, a set of requirements was established and some initial assumptions were made as to the geometry and operating parameters of this motor. The initial assumptions and performance requirements are summarized in Table 1.

Table 1. Motor Specifications

Number of Phases	3
Frequency (Hz)	60
Rotor speed (rpm)	200
Rated power (Hp)	40000
Operating voltage (V)	1000
Power factor	0.8
Winding factor, k_w	0.9
L/D	0.22
Tooth fraction, λ_p	0.5

A line frequency of 60 Hz was selected for the motor design since this is the frequency commonly used in U. S. Navy ships electric service applications.

The type of machine that was used in this research is a flux concentrating permanent magnet machine. In this type of machine the magnets are oriented so that their magnetization is azimuthal as shown in figure 2-1.

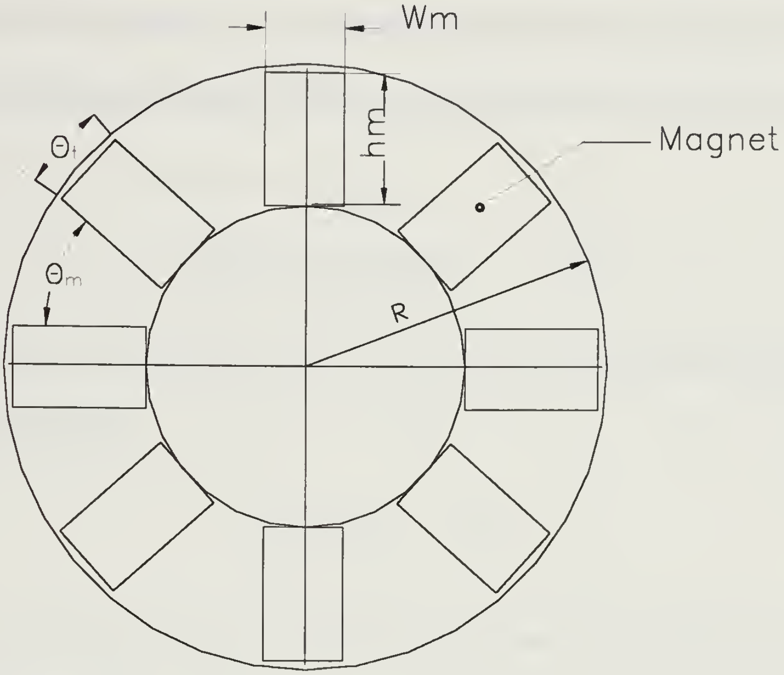

Figure 2-1. Motor cross section

Using the geometry described in figure 2-1 and the requirements specified in table 1, some of the initial sizing considerations were started. For the required rotor speed, n, and electrical frequency, f, the number of pole pairs, p, in the machine was determined using $p = 60f/n$. A rotor speed of 200 rpm was selected for this design because this speed is consistent with rated shaft speeds for naval ships.

Since this machine will be used for ship propulsion, one of its desired attributes is that it is comparable is size and weight to conventional propulsion machinery. By calculating the radius and the length of the machine, an initial determination of the

feasibility of the motor can be obtained. Weight calculations are performed once all the machine components are sized.

Once the rated speed of the rotor was established, the radius of the rotor, R, was determined from the mechanical power output required and the average gap shear stress, $\langle\tau\rangle$. To calculate the radius of the rotor the following two equations were used

$$\text{Power} = (\text{Torque})\omega_m \qquad (2\text{-}1)$$

$$\text{Torque} = \langle\tau\rangle(2\pi RL)R \qquad (2\text{-}2)$$

The aspect ratio indicated in table 1 was calculated using the following relation suggested by Levi in [10]

$$\frac{L}{D} \approx \frac{\pi}{2} p^{-\frac{2}{3}} \qquad (2\text{-}3)$$

The aspect ratio estimated using equation (2-3) minimizes the winding resistance for a given electromotive force.

Using the aspect ratio, L/D, calculated from (2-3) and combining equations (2-1) and (2-2), the radius of the rotor is

$$R = \sqrt[3]{\frac{\text{Power}}{4\pi\omega_m\langle\tau\rangle(L/D)}} \qquad (2\text{-}4)$$

Once the machine radius was calculated, the active length of the machine was determined as $L = 2R(L/D)$.

A value for the average gap shear was estimated in order to calculate the size of the machine. The average gap shear stress can be estimated by $\langle\tau\rangle \approx \frac{1}{\sqrt{2}} B_1 K$, where B_1 is the peak value of the fundamental magnetic flux density and K is the root mean square

(rms) value of the effective armature reaction current density [6]. Gap shear has units of pressure, pascals, (Pa).

For this machine design the value selected for the average gap shear was based on machine sizing considerations and cooling requirements. By maximizing the gap shear, the machine radius can be reduced, equation (2-4); however, high values of shear will require additional cooling requirements.

The value of shear stress selected is that of an air-cooled machine. This type of machine was selected in an effort to maintain simplicity. For this type of machine, and for the specified rating, a shear value of approximately 50,000 Pa was obtained [6].

Figure 2-2 shows the variation of rotor radius as a function of gap shear. From equation (2-4) it can be noted that $R \propto \frac{1}{\sqrt[3]{\tau}}$. Higher shear values will result in smaller machines. However, these smaller machines will require additional cooling provisions, adding to the complexity and most likely to the overall size of the machine.

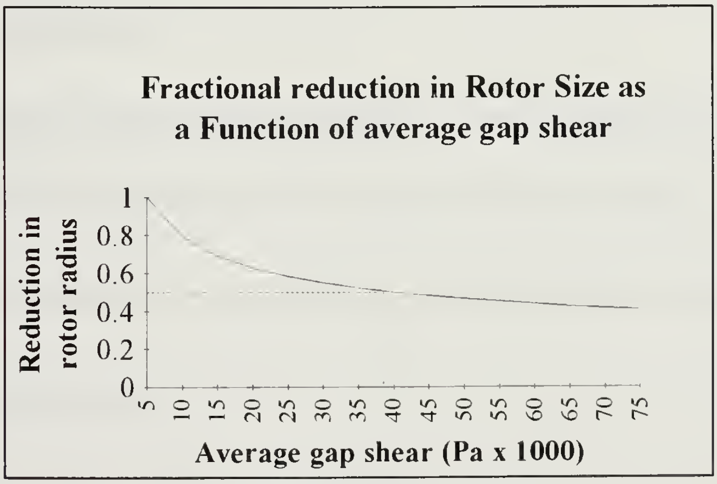

Figure 2-2. Radius reduction as a function of gap shear

Considering the above reasons an average gap shear of 50,000 Pa was selected and used for the motor design. This shear value is at the high end for typical slow speed air-cooled machines [6]. For the specified shear and aspect ratio the rotor diameter was calculated to be approximately 4.2 m and the length of the machine, L, approximately one meter.

2.1 Air gap size

Mechanical considerations and performance requirements drive the length of the air gap, g. The minimum physical length, in meters, of the air gap can be estimated from the following relation [11]

$$g \approx 3.35 \times 10^{-3}\left(\frac{L}{D}\right)^{3/4} D \qquad (2\text{-}5)$$

Using the calculated radius of the rotor, equation (2-4), and L/D, (2-3) a minimum gap length of 5 mm is obtained. This gap length was used for this research.

2.2 Magnet dimensions

Considering the geometry shown in figure 2-1, and assuming that the magnets occupy one half of the circumference of the rotor, the width of a magnet is given by $w_m = \frac{\pi R}{p}$ and from figure 2-1 $w_m = 2R\sin\frac{\theta_t}{2}$. Combining these two equations and solving for the magnet angle, θ_t

$$\theta_t = 2\sin^{-1}\frac{\pi}{2p} \qquad (2\text{-}6)$$

Then the pole face angle, θ_m, is given by

$$\theta_m = \frac{\pi}{p} - \theta_t \tag{2-7}$$

The remaining dimension of the magnets yet to be determined is the height, h_m. This dimension is determined by considerations other than just geometry, such as the magnetic flux density at the air gap. In order to calculate the magnet height, the gap flux density needs to be determined.

In flux concentrating machines, the flux density in the gap is greater than the flux density in the magnets. To calculate the fields in this machine a simple reluctance model presented in [12] was used. For the given geometry, the flux path in the rotor involves a magnet and one half of each of two adjacent pole pieces. For one pole piece the permeance of the air gap is given by

$$\wp_{gap} = \mu_o L \frac{R\theta_m}{g} \tag{2-8}$$

and the permeance of a magnet is

$$\wp_m = \mu_o L \frac{h_m}{w_m} \tag{2-9}$$

The magnetic flux density in the magnets, B_m, is calculated using a simplified magnetic circuit. This circuit consists of the magnet's own permeance in series with one half of the permeance of each of two pole pieces. So that the flux density of the magnets is given by

$$B_m = B_o \frac{\wp_{gap}}{\wp_{gap} + \wp_m} \tag{2-10}$$

and the flux density in the gap can be calculated

$$B_{gap} = B_m \frac{2\,h_m}{R\,\theta_m} \qquad (2\text{-}11)$$

Solving equation (2-11) for the magnet height, the following relation is obtained

$$h_m = \frac{B_{gap}}{B_m} \frac{R}{2} \theta_m \qquad (2\text{-}12)$$

For a given radius and magnet angle, the ratio of gap flux to magnet flux will determine the magnet height. The gap flux density that can be achieved in the machine will be limited by stator teeth and back iron saturation. This limit will be checked later in the design. Once a magnet height is selected, the magnet spacing needs to be checked. For the given geometry the closest point between two magnets occurs at the interior corners. Using simple geometry the distance between two adjacent corners can be shown to be [12]

$$s = 2\left(R - h_m\right) \sin\frac{\pi}{2p} - w_m \cos\frac{\pi}{2p} \qquad (2\text{-}13)$$

After checking that the magnet dimensions are compatible with the rotor size and the gap flux density has been selected, the surface current density, K, necessary for operation of the machine at rated power can be calculated, $K \approx \frac{\sqrt{2}\langle\tau\rangle}{B_1}$.

2.3 Determination of terminal current and machine rating

For a machine with a small gap, it can be assumed that the magnetic flux is not a function of radial position. The space fundamental of this flux is then of the form

$$B_1 = \frac{4}{\pi} \sin\frac{p\theta_m}{2} B_{gap} \qquad (2\text{-}14)$$

With this value of flux, the internal voltage can be estimated using

$$E_{af} = \frac{2RLN_s k_s \omega}{p} \frac{B_1}{\sqrt{2}} \qquad (2\text{-}15)$$

The terminal current into the machine, I_t, is given by

$$I_t = \frac{N_{slots} w_{slots} h_{slot}}{6N_s} J_a \qquad (2\text{-}16)$$

where $N_{slots} w_{slots} = 2\pi R \lambda_p$ and $K = J_a h_{slot} k_s$. Once the internal voltage and the terminal current are known, the rating of the machine $|P + jQ| = 3V_t I_t$ can be determined if the ratio of terminal voltage to internal voltage, $v = V_t / E_{af}$, is known. A method to calculate v will be proposed later in the chapter; however, this method requires that the machine reactances be known.

2.4 Machine Reactances

The permanent magnets are in the main flux path of the armature. The presence of the magnets will make the machine salient since the direct and quadrature axes will be affected differently. Derivation of the direct and quadrature axes' reactances is shown in several places; however, for consistency the notation used in [12] will be maintained. The direct and quadrature reactances, X_d and X_q, are given by

$$X_d = \frac{3}{2} \frac{4}{\pi} \frac{\mu_o N_s^2 k_s^2 RL}{p^2 g} \omega \gamma \sin \frac{p\theta_m}{2} \qquad (2\text{-}17)$$

$$X_q = \frac{3}{2} \frac{4}{\pi} \frac{\mu_o N_s^2 k_s^2 RL}{p^2 g} \omega \left(1 - \sin \frac{p\theta_t}{2}\right) \qquad (2\text{-}18)$$

These formulas have a new unknown, the number of turns per phase in the stator, N_S. The value of N_S will be calculated once v and the rating of the machine are known.

2.5 Internal Voltage

To calculate the ratio of terminal voltage to internal voltage the phasor diagram shown in figure 2-3 is used. A two-axis representation of the machine is shown in figure 2-3. The mathematical transformation used to derive the two-axis model is discussed in chapter three.

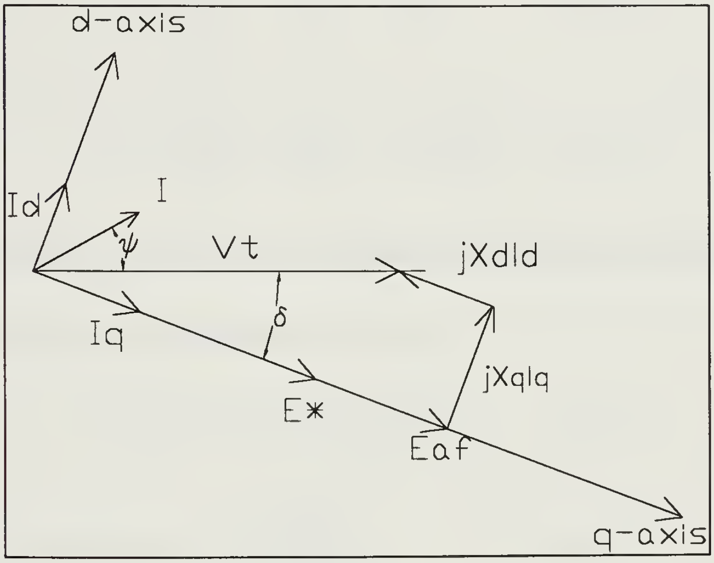

Figure 2-3. Phasor diagram for Negatively Salient Motor

From figure 2-3 the following set of phasor relations are derived

$$E^* = \sqrt{V_t^2 + \left(I_t X_q\right)^2 - 2V_t I X_q \sin\psi} \qquad (2\text{-}19)$$

$$I_d = I_t \sin(\psi + \delta) \qquad (2\text{-}20)$$

$$\delta = \tan^{-1} \frac{IX_q \cos\psi}{V + IX_q \cos\psi} \qquad (2\text{-}21)$$

$$E_{af} = E^* - \left(X_q - X_d\right)I_d \tag{2-22}$$

To solve this set of equations an iterative method is suggested in [12]. The end result of the proposed method is a value for the ratio of terminal to internal voltage, v. To implement the method, the set of equations (2-19) through (2-22) and the machine reactances, equations (2-17) and (2-18) are normalized with respect to the internal voltage, equation (2-13). After normalizing, per unit scaling, and some simplification the reactances are given by

$$x_{ad} = \frac{\mu_o R}{pg}\lambda_p \sqrt{2}\frac{K}{B_1}\gamma \sin\frac{p\theta_m}{2} \tag{2-23}$$

$$x_{aq} = \frac{\mu_o R}{pg}\lambda_p \sqrt{2}\frac{K}{B_1}\left(1 - \sin\frac{p\theta_t}{2}\right) \tag{2-24}$$

The set of phasor relations after normalizing and assuming operation at rated current ($i_t = 1.0$ p.u.) and power factor is given by

$$e^* = \sqrt{v^2 + x_{aq}^2 - 2x_{aq} v \sin\psi} \tag{2-25}$$

$$\delta = \tan^{-1}\frac{x_{aq} v \cos\psi}{v + x_{aq} v \sin\psi} \tag{2-26}$$

$$i_d = \sin(\psi + \delta) \tag{2-27}$$

$$e_{af} = e^* - (x_{aq} - x_{ad})i_d \tag{2-28}$$

Using the normalized reactances and the set of equations (2-25) through (2-28) the suggested method of [12] seeks a set of values that will solve the above set of equations in which the fixed point is $e_{af} = 1.0$. The iterative method is started by selecting an arbitrary value for v and solving equations (2-25) through (2-28) sequentially. For subsequent

iterations a new value of $v_{new} = v_{old}/\sqrt{e_{af}}$ is used and the iterative process is continued until e_{af} converges to 1.0. Once convergence is obtained the final value of v is used to determine the machine rating

$$3V_tI_t = \frac{2\pi}{\sqrt{2}}\frac{\omega}{p}R^2L\lambda_pKB_1v \quad (2\text{-}29)$$

If a value of terminal voltage is selected, table 1, then the machine terminal current can be calculated, equation (2-15).

2.6 Stator sizing

Once the terminal current is known the number of stator turns per phase is calculated, equation (2-16). The slot width and height, the number of slots and the size of the back iron need to be calculated to complete the initial sizing of the machine. If a tooth fraction of $\lambda_p = 0.5$ is used and assuming that $R >> g$, then the slot width is calculated as

$$w_s = \frac{2\pi R}{6N_sk_s} \quad (2\text{-}30)$$

and the number of slots in the stator is given by

$$N_{slots} = 6N_sk_s\lambda_p \qquad (2\text{-}31)$$

The slot height, h_s, is determined principally by the insulation required by the armature windings. Commonly these requirements are such that the copper area in the slot is between 40% to 60% of the area slot [6]. For this design the slot copper fraction, k_{Cu}, selected was 0.5. With these assumptions the slot height is calculated by

$$h_{slot} = \frac{K}{J_ak_sk_{Cu}} \qquad (2\text{-}32)$$

Limits on the current density, J_a, are established by the maximum temperature rise, θ_w, in degrees Kelvin, above ambient temperature (40 C), allowed in the copper. The allowable temperature rise is based on the class of insulation used in the machine. For an air-cooled machine, the maximum current density based on thermal considerations can be estimated by

$$J_a \leq \frac{h\theta_w\gamma_{Cu}k_s}{K} \qquad (2\text{-}33)$$

where γ_{Cu} is the conductivity of copper and h is the overall heat transfer coefficient. Equation (2-33) represents the energy balance between the heat generated in the copper and the heat removed by the cooling medium, air [11]. In equation (2-33) an overall heat transfer coefficient, $h = 30\frac{\text{watts}}{^\circ\text{K} \times \text{meter}^2}$ was used. This coefficient was derived for an air-cooled machine and is based on empirical calculations discussed in [11].

Equations (2-1) through (2-32) are used in Appendix A to do the initial machine sizing. Based on the dimensions calculated by these equations and the established machine geometry, the weight of the machine was estimated. In addition, once the dimensions and geometry of the machine have been estimated, other machine parameters such as the machine efficiency and stator leakage reactance can be estimated.

2.7 Stator leakage reactance

The stator leakage reactance was calculated using the methods of reference [11]. Leakage reactance consists of (1) slot, (2) tooth top, (3) end winding, and (4) harmonic reactances. Tooth top leakage reactance is proportional to the gap length. In

synchronous machines with permanent magnet field the gap length is small and tooth top leakage reactance can be neglected [11].

For an initial estimate, the leakage reactance was assumed to consist of slot and harmonic leakage reactances. The slot leakage and harmonic reactances were calculated using the procedure outlined in [11]. The calculated reactance was increased by 10% to account for the effects of end winding reactance in the total leakage reactance.

The leakage reactance was assumed to be the same for the direct and quadrature axes. This assumption is commonly made in the literature [11].

2.8 Losses and machine efficiency

The power dissipated in the machine determines its efficiency. This dissipated power will determine the cooling and ventilation requirements for the machine to ensure that allowable temperature limits are not exceeded. The losses considered in Appendix A comprise: no-load losses in the iron, friction and windage losses, copper losses and load losses.

The rotating magnetic field will result in heat losses in the iron due to eddy currents and hysteresis. These type of losses occur primarily in the stator teeth, the surface of the rotor poles and the structural parts of the machine exposed to alternating magnetic fields. These losses are proportional to the square of the peak value of B. Hysteresis losses are proportional to the frequency and eddy current losses are proportional to the square of the frequency. Core losses are estimated in Appendix A using the relations discussed in [11].

Mechanical losses in the machine are the result of friction in the bearings and windage between the rotor and the stator. These losses were estimated using the relation presented in [11].

The copper losses were calculated by estimating the resistance of the armature windings and using the rated terminal current. To calculate the armature resistance, the mean length of conductor per phase in the armature was estimated as suggested in [11]. For a three phase machine the armature copper losses are $3R_aI^2$, where R_a is the armature winding resistance and I is the line current.

Load losses are caused by the flux produced by the armature currents. These losses include eddy current losses in the support structures, pole surfaces and damper windings. The load losses were estimated as 1% of the armature copper losses [10].

The machine efficiency was estimated in Appendix A using

$$\text{Efficiency} = 1 - \frac{\text{losses}}{\text{input}}$$

where the losses are the sum of the individual losses described in the preceding paragraphs. The calculated machine efficiency was 98%, close to the efficiency predicted by [6].

2.9 Back iron sizing

The thickness of the back iron, h_c, must be sufficient to carry the machine flux. Assuming a sinusoidal air gap flux density, B_{gap}, the back iron thickness is determined using

$$\oiint_S \vec{B} \cdot d\vec{S} = 0 \qquad (2\text{-}34)$$

If the air gap flux is assumed to be constant, using the specified machine geometry and if R>>g, equation (2-34) can be simplified to

$$h_c = \left(\frac{B_{gap}}{B_s}\right)\left(\frac{R}{p}\right) \qquad (2\text{-}35)$$

where B_S represents the flux density in the back iron.

The minimum back iron thickness is then calculated by using a back iron flux density close to saturation. A value of B_S = 1.2 T, rms, was used for the machine design.

2.10 Weight Calculations

A machine weight was estimated by calculating a rotor weight and a stator weight for the given machine geometry. The material composition of the different components was assumed to be that of use in standard motor construction. The machine weight calculated does not include weight of foundations, the weight of a cooling system or the weight of an enclosure for the motor.

The weight of the motor is expressed in long tons[1] (lton) for easier comparison with current or conceptual drives for naval ships.

---

[1] One long ton = 1016 kilograms

Chapter 3. Dynamic Models

To perform the fault simulations a model of the permanent magnet motor and of the electrical fault were derived. These two models were incorporated into a dynamic simulation model developed by Professor James L. Kirtley, MIT. This chapter describes the various models used in the simulation. Models for a permanent magnet ac synchronous motor, arc fault and interconnections between the motor and the power system are presented.

3.1 Two axis transformation

The permanent magnet motor model used in this research is based on the derivations presented in [11] through [14]. It assumes linear magnetics and sinusoidal stator winding distributions. Inductance and resistance values for the motor were calculated using the methods discussed in chapter two and Appendix A.

Using the symmetry of cylindrical rotation, a coordinate transformation that accounts for the relative motion between the stator and the rotor is introduced. This transformation maps the stator winding variables to a reference frame that rotates with the rotor. In this reference frame, mutual inductances are independent of rotor position. Such transformation is commonly known as the Park's transformation. A version of this transformation used in [5] is given by

$$T = \sqrt{\frac{2}{3}}\begin{bmatrix} \cos\theta & \cos\left(\theta - \frac{2\pi}{3}\right) & \cos\left(\theta + \frac{2\pi}{3}\right) \\ -\sin\theta & -\sin\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \qquad (3\text{-}1)$$

where θ is some arbitrary angle. For a reference frame rotating with the rotor $\theta = \omega t$ where ω is the speed of the rotor. This transformation is orthogonal and power invariant, regardless of the choice of angle. The inverse transformation is given by

$$T^{-1} = \sqrt{\frac{2}{3}}\begin{bmatrix} \cos\theta & -\sin\theta & \frac{1}{\sqrt{2}} \\ \cos\left(\theta - \frac{2\pi}{3}\right) & -\sin\left(\theta - \frac{2\pi}{3}\right) & \frac{1}{\sqrt{2}} \\ \cos\left(\theta + \frac{2\pi}{3}\right) & -\sin\left(\theta + \frac{2\pi}{3}\right) & \frac{1}{\sqrt{2}} \end{bmatrix} \qquad (3\text{-}2)$$

If the Park's transformation is applied to an arbitrary vector, **F**, representing a set of stator variables, such as currents or voltages, the stator variables are mapped to a fixed rotor reference frame. In this stationary reference frame the machine reactances are constant, independent of rotor position.

$$\begin{aligned} F_{dq0} &= TF_{abc} \\ F_{abc} &= T^{-1}F_{dq0} \end{aligned} \qquad (3\text{-}3)$$

The d and q subscripts are used to represent the direct and quadrature axes, respectively. The direct axis is aligned with the polar axis and the quadrature axis is aligned with the interpolar space. Zero-sequence variables are represented by the subscript 0 in

equations (3-3). The zero-sequence components represent components of armature currents that do not produce a net air gap flux [4].

3.2 Per Unit Scaling

The models used in the simulation have been scaled to a per unit (p.u.) system. Scaling to a per unit system produces variables whose magnitudes are close to one. The base values selected for this scaling can be arbitrary. However, for this analysis it is convenient to use the motor's rated power and voltage as the base values for the per unit scaling. Expressing the motor variables in a per unit system allows the comparison of this motor against other motors regardless of their ratings.

For this motor the base quantities are defined by

$$\begin{aligned} P_B &= \frac{3}{2} V_B I_B \\ Z_B &= \frac{V_B}{I_B} \qquad (3\text{-}4) \\ \lambda_B &= \frac{V_B}{\omega_o} \end{aligned}$$

To convert the motor parameters to the per unit system, the ordinary variables were divided by the corresponding base quantities. A new quantity will be introduced into the model to change torque to the per unit system. This new constant is defined as the inertia constant, H; it represents the rotor's kinetic energy at rated speed divided by the motor rated power. The inertia constant, H, has units of seconds and is expressed as

$$H = \frac{\frac{1}{2} J\omega_m^2}{P_B} \qquad (3\text{-}5)$$

In equation (3-5), J is the moment of inertia of the shaft and ω_m is the shaft's rotational speed. In general the moment of inertia of a circular shaft can be calculated as $J = \int_V r^2 \rho dV$, where ρ is the shaft's material density [3]. It can be shown that for a circular shaft this is equivalent to $J = Wk^2$, where W is the mass of the shaft and k is its radius of gyration. For a solid circular shaft of radius, R, the radius of gyration is given by $k = R/\sqrt{2}$. A value for H was calculated for this motor using the mass and rotor radius calculated in Appendix A.

Since the rotor is directly connected to the propulsion shaft, the length of the shaft and the propeller needs to be included in the moment of inertia calculation. The weight of the propeller and of the propulsion shafts can be estimated from propeller and shaft weights from U.S. Navy ships or commercial ships.

3.3 Permanent Magnet Motor Model

The permanent magnet machine model is defined by a set of six coupled electro-mechanical equations. The electrical equations in this set are statements of Kirchoff and Faraday's laws, which describe the voltage-current relations in the machine. The mechanical equations are statements of Newton's Second Law of Motion. Coupling between the mechanical and electrical systems is through the dependence of electrical torque on flux linkages and through the dependence of flux linkages on torque angle, δ.

The machine parameters used for the simulations are calculated in Appendix A using the methodology discussed in chapter two. A two-axis (d-q axis) representation of the synchronous machine is used.

In the d-q reference frame, the direct axis of the motor is represented by the following circuit based on the model presented in references [5] and [12].

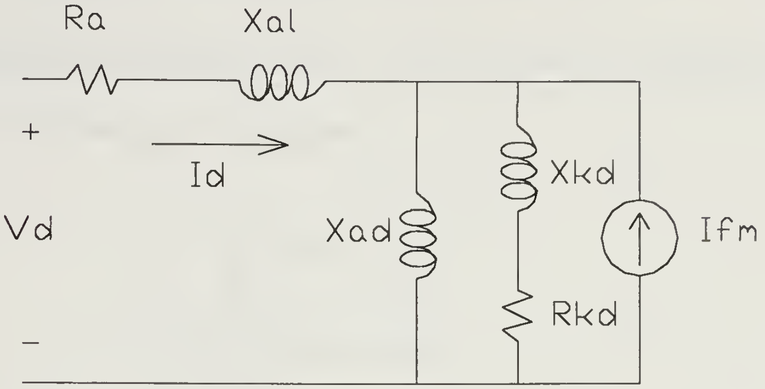

Figure 3-1. Direct axis circuit representation

where the constant current source represents the field generated by the permanent magnets.

Similarly the quadrature axis of the motor can be represented by the following circuit.

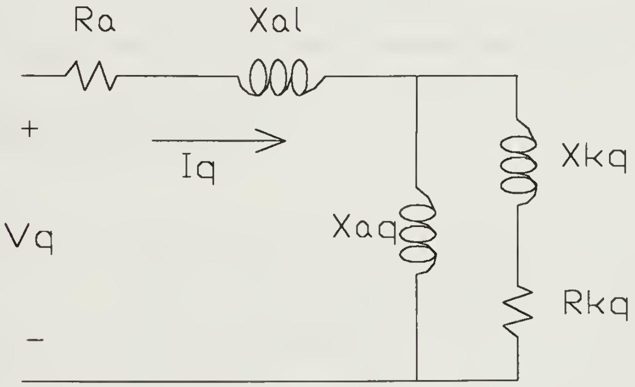

Figure 3-2. Quadrature axis circuit representation

The following notation has been introduced in the above two models:

X_{ad} and X_{aq} represent the magnetizing reactances,

X_{al} represents the armature leakage reactance, which is assumed to be equal for the d and q axes,

R_a is the armature resistance,

X_{kd} and X_{kq} are the damper winding reactances, and

R_{kd} and R_{kq} are the damper winding resistances.

Using the models shown in figures 3-2 and 3-3, the three phase ac synchronous machine is represented by the following set of equations

$$\begin{aligned}
V_d &= \frac{d\lambda_d}{dt} + R_a I_d - \lambda_q \omega \\
V_q &= \frac{d\lambda_q}{dt} + R_a I_q + \lambda_d \omega \\
V_{kd} &= \frac{d\lambda_{kd}}{dt} + R_{kd} I_{kd} \\
V_{kq} &= \frac{d\lambda_{kq}}{dt} + R_{kq} I_{kq} \\
T_e &= \frac{3}{2}(\lambda_d I_q - \lambda_q I_d)\frac{\omega_o}{\omega_m}
\end{aligned} \tag{3-6}$$

For permanent magnet excitation, the field is represented by the constant current source , I_{fm}.

The flux equations for the machine are

$$\begin{aligned}
\lambda_d &= L_d I_d + L_{ad} I_{kd} + L_{ad} I_{fm} \\
\lambda_{kd} &= L_{ad} I_d + L_{kd} I_{kd} + L_{ad} I_{fm} \\
\lambda_q &= L_q I_q + L_{aq} I_{kq} \\
\lambda_{kq} &= L_{aq} I_q + L_{kq} I_{kq}
\end{aligned} \tag{3-7}$$

where the inductances, L_d and L_q, are defined by

$$L_d = L_{al} + L_{ad}$$

$$L_q = L_{al} + L_{aq}$$

The simulation code was written to be used with any ac synchronous machine, regardless of its rating. For this reason it is intended to be used with machines scaled in the per

unit system. To use the dynamic simulation code, the above set of equations were scaled in the per unit system. Dividing by the appropriate base quantities, the sets of equations (3-5) and (3-6) are represented in the per unit system by

$$\begin{bmatrix} \psi_d \\ \psi_{kd} \end{bmatrix} = \begin{bmatrix} x_d & x_{ad} & x_{ad} \\ x_{ad} & x_{kd} & x_{ad} \end{bmatrix} \begin{bmatrix} i_d \\ i_{kd} \\ i_{fm} \end{bmatrix} \tag{3-8}$$

$$\begin{bmatrix} \psi_q \\ \psi_{kq} \end{bmatrix} = \begin{bmatrix} x_q & x_{aq} \\ x_{aq} & x_{kq} \end{bmatrix} \begin{bmatrix} i_q \\ i_{kq} \end{bmatrix} \tag{3-9}$$

$$\begin{aligned} v_d &= \frac{1}{\omega_o}\frac{d\psi_d}{dt} + r_a i_d - \psi_q \frac{\omega}{\omega_o} \\ v_q &= \frac{1}{\omega_o}\frac{d\psi_q}{dt} + r_a i_q + \psi_d \frac{\omega}{\omega_o} \\ 0 &= \frac{1}{\omega_o}\frac{d\psi_{kd}}{dt} + r_{kd} i_{kd} \\ 0 &= \frac{1}{\omega_o}\frac{d\psi_{kq}}{dt} + r_{kq} i_{kq} \\ T_e &= (\psi_d i_q - \psi_q i_d) \end{aligned} \tag{3-10}$$

The damper windings' voltages, v_{kd} and v_{kq}, were set to zero since these windings are normally shorted. Using a Thevenin equivalent circuit of the d-axis model, figure 3-1, the constant current source can be represented as a voltage source, $E_{af} = X_{ad} I_{fm}$ in series with the magnetizing impedance, X_{ad}.

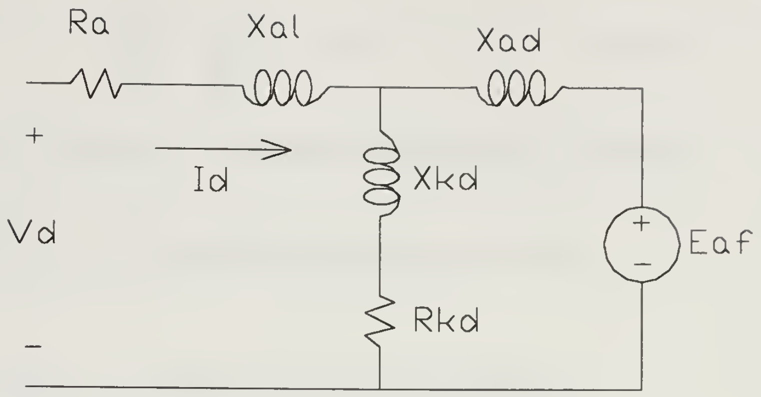

Figure 3-3. Thevenin equivalent of d-axis model

With the model presented in figure 3-3, equation (3-7) can be rewritten as follows

$$\begin{bmatrix} \psi_d \\ \psi_{kd} \end{bmatrix} = \begin{bmatrix} x_d & x_{ad} \\ x_{ad} & x_{kd} \end{bmatrix} \begin{bmatrix} i_d \\ i_{kd} \end{bmatrix} + \begin{bmatrix} e_{af} \\ e_{af} \end{bmatrix} \qquad (3\text{-}11)$$

By solving the above equations for the currents, the d-q model of the motor can then be incorporated into a simulation code that connects the motor to an external network. Solving equations (3-9) through (3-11) for the currents, i_d and i_q, and after some simplifications the following sixth order state-space representation of the motor was obtained

$$\frac{d\psi_d}{dt} = -\frac{\psi_d}{T_{ad}} + \frac{e_q''}{T_{ad}} + \omega\psi_q + \omega_o v_d \qquad (3\text{-}12)$$

$$\frac{d\psi_q}{dt} = -\frac{\psi_q}{T_{aq}} - \frac{e_d''}{T_{aq}} - \omega\psi_d + \omega_o v_q \qquad (3\text{-}13)$$

$$\frac{de_q''}{dt} = -\frac{e_q''}{T_{do}''} + \frac{(x_d - x_d'')}{T_{do}''} i_d + \frac{e_{af}}{T_{do}''} \qquad (3\text{-}14)$$

$$\frac{de_d''}{dt} = -\frac{e_d''}{T_{qo}''} - \frac{(x_q - x_q'')}{T_{qo}''} i_q \qquad (3\text{-}15)$$

$$\frac{d\delta}{dt} = \omega - \omega_o \qquad (3\text{-}16)$$

$$\frac{d\omega}{dt} = \frac{\omega_o}{2H}(T_m - T_e) \qquad (3\text{-}17)$$

In equations (3-12) through (3-17) the following quantities are defined:

$e''_q = \frac{x_{ad}}{x_{kd}}\psi_{kd}$ = Voltage behind subtransient reactance,

$e''_d = -\frac{x_{aq}}{x_q}\psi_{kq}$ = Voltage behind subtransient reactance,

$T''_{do} = \frac{x_{kd}}{\omega_o r_{kd}}$ = D-axis open circuit subtransient time constant,

$T''_{qo} = \frac{x_{kd}}{\omega_o r_{kd}}$ = Q-axis open circuit subtransient time constant,

$T_{ad} = \frac{x''_d}{\omega_o r_a}$ = Direct axis armature time constant,

$T_{aq} = \frac{x''_q}{\omega_o r_a}$ = Quadrature axis time constant,

$x''_d = x_d - \frac{x^2_{ad}}{x_{kd}}$ = D-axis subtransient reactance,

$x''_q = x_q - \frac{x^2_{aq}}{x_{kq}}$ = Q-axis subtransient reactance,

The stator currents, in the motor reference frame, have been defined by

$$i_d = \frac{\psi_d - e''_q}{x''_d} \qquad (3\text{-}18)$$

$$i_q = \frac{\psi_q + e''_d}{x''_q} \qquad (3\text{-}19)$$

Equations (3-12) and (3-13) are the stator equations, which have time constants of the order of $1/\omega_o = 0.0026$ seconds. If the transients of interest are in the order of 0.1 to several seconds then two stator equations, equations (3-12) and (3-13), will become algebraic equations provided that the following conditions hold

$$\omega >> \frac{1}{T_{ad}}, \frac{1}{T_{aq}}, \frac{d}{dt}$$

Furthermore if $\omega \approx \omega_o$ then equations (3-12) and (3-13) simplify to

$$\psi_d = v_d \qquad (3\text{-}12a)$$

$$\psi_q = v_q \qquad (3\text{-}13a)$$

With these two algebraic equations the machine model has been reduced to a fourth order model, equations (3-14) through (3-17).

3.4 Network model

The external network model used in the simulation code is based on the model derived in [15]. This model uses the machine currents to interface with the network. The definition of the network is accomplished by defining the nodes and branches of the network. This model is discussed in chapter four.

3.5 Fault Model

The fault model used for this research is an ac arc fault whose properties and characteristics are described in detail in [2]. Reference [2] describes the electric arc as a self-sustained electrical discharge having a low voltage drop and capable of supporting large

currents. At atmospheric pressure the temperature of the arc will reach temperatures as high as 6,000K.

According to [2], for large currents the arc voltage, e_a, is given by

$$e_a = c_1 i + \frac{c_2}{i} + c_3 i^2 \qquad (3\text{-}20)$$

where the constants c_1 through c_3 are determined by the electrode materials. The voltage-ampere characteristic of ac arcs will show hysteresis effects with distinct ignition and extinction voltages [2].

The arc is modeled as a constant voltage drop; for calculation simplifications the values of ignition and extinction voltages will be neglected. For the materials considered such as copper and iron the arc voltage drop is approximately 60 volts [2]. The voltage-current relation of the fault can then be approximated by a characteristic such as that shown in figure 3-4.

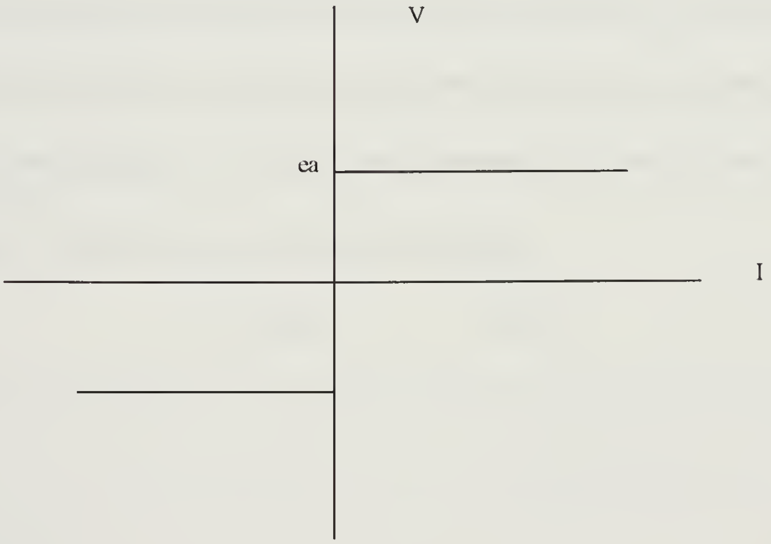

Figure 3-4. Fault voltage-current characteristic

For the purposes of this research the impedance of the fault is assumed to be much smaller than the synchronous impedance of the motor. This is simulated by using fault impedance values ten times lower than the synchronous impedance values for the motor.

The power dissipated by the fault is estimated by multiplying the fault current by the fault voltage.

3.6 Motor Load Model

The mechanical load placed on the motor by the ship is represented by a mechanical torque on the shaft. The power delivered by a propulsion shaft for a ship moving at high speed is proportional to the speed of the ship cubed, $P \propto v^3$ [16]. This assumption is not valid for ships such as destroyers that can reach speeds near or above hull speed[2]. At speeds close to the hull speed, the power required may vary with speed raised to a power approaching 6 or 7 [16].

Once a propeller has been selected and matched to the hull, it can be shown the rotational speed of the shaft is proportional to the speed of the ship. Using this relation, and since $P = T\omega_m$ then the torque on the shaft can be assumed to be proportional to the square of the shaft speed. This torque is assumed to be of the form

$$T_m = a\omega_m^2 \qquad (3\text{-}21)$$

---

[2] Hull speed or critical speed length ratio is the speed at which wave making resistance starts accumulating most rapidly. Hull speed is calculated as $V_{hull}(\text{knots}) = 1.34\sqrt{L_s}$, where L_s is the ship's length in feet.

The constant is calculated with the motor operating at full load using the rated shaft speed and torque calculated in Appendix A. It is further assumed that this constant is independent of shaft speed.

A windmilling shaft is not delivering any thrust or torque; however, it will rotate at a speed determined by the characteristics of the propeller and the speed of the ship. Once the propeller is selected, for a fixed pitch propeller, the speed of rotation of the shaft, while windmilling, will be proportional to the speed of the ship. So as the ship slows down the windmilling shaft will slow down proportionately. To incorporate this model into the dynamic simulation model it is necessary to determine how the ship's speed changes as a function of time while the ship is coasting down.

For a ship moving on a constant course at some initial speed, v_0, the total kinetic energy of the ship, U, is of the form $U = \frac{1}{2}Mv_o^2$. The symbol M accounts for the mass of the ship and the "added" mass in the direction of motion. While a ship coasts down from some given speed the rate at which the ship loses energy to the water is equal to the effective horsepower, EHP, of the ship

$$\frac{dU}{dt} = -EHP \qquad (3\text{-}22)$$

In general EHP is proportional to speed elevated to some power, $EHP = kv^n$, where k is some constant. During a short time interval, $\Delta t = t_o - t$, equation (3-22) can be written after some manipulations as

$$\frac{1}{2}M(v_o^2 - v^2) = -(t_o - t)(kv^n) \qquad (3\text{-}23)$$

If the ship is assumed to be moving at an initial speed, v_o, at time, $t_o = 0$, then from (3-23)

$$v_o^2 - v^2 = \frac{2kv^n}{M}t \qquad (3\text{-}24)$$

For speeds close to the initial ship speed, $v \approx v_o$, equation (3-24) can be rewritten as

$$2v \approx \frac{2kv^n}{M}t \qquad (3\text{-}25)$$

Using equation (3-25) the following relation between ship's speed and time is obtained

$$v \propto \frac{1}{\sqrt[n-1]{t}} \qquad (3\text{-}26)$$

Since the rotational speed of a windmilling shaft is proportional to the speed of the ship, using equation (3-26) the rotational speed of the shaft, ω_m, will exhibit the same time dependence as the ship's speed.

If the ship is operating below hull speed, which is the case for most full form ships, then at its rated speed it can be assumed that $EHP \propto v^3$. For n=3, then $v \propto \frac{1}{\sqrt{t}}$. After the casualty takes place and the motor main supply breakers are opened, the shaft will continue to rotate at a speed proportional to the speed of the ship. It can be assumed that the increase in ship's resistance added by the windmilling shaft is negligible compared to the hull resistance [17].

Considering the discussion above, while the ship is coasting down, the rotational speed of the shaft was assumed to be of the form $\omega_m \propto \frac{1}{\sqrt{t}}$. The final shaft speed will be proportional to the final ship speed.

Chapter 4. Simulation Model

With the system model complete, as discussed in chapter three, the dynamic response of the permanent magnet motor and the internal fault was investigated. The purpose of conducting these studies was to determine the effect of the internal fault after the motor is disconnected from the system. The possible damage done by the fault will occur in a very short time so that the time delay of the relay and protection system might not be significant.

A possible way to reduce the effects of the fault is to short circuit the motor terminals after the motor is disconnected from its power supply. By shorting the motor terminals it might be possible to reduce the power input into the fault, thus serving to minimize the effects of the fault.

4.1 Discussion of the simulation program

The simulation code was written in C++. Using this language allowed the use of object-oriented programming (OOP). This type of programming consists of building programs as a collection of abstract data type instances. Operations performed on the object types are the abstract operations that solve the problem. These objects serve as modules that can be reused for solving another problem in the same domain [17]. In the simulation code generators, motors, nodes and lines are built into objects. These objects are invoked throughout the code as needed to solve the simulation problem.

The dynamic simulation program, Appendices B through E, requires the user to provide four input files. The first two files are used to define the node types and the interconnection lines between nodes and their reactances. In addition, they define the nodes to which the generators or motors are connected and the names of the output files that will contain the node voltages and line currents.

Two other input files are needed for the simulation. One file defines the electrical parameters of the machine. The other file defines the timing and sequence of events for the simulation. The machine's electrical parameters used in the simulation were those calculated using Appendix A.

The simulation code consists of four executable programs that can be run individually or using a shell program that executes all four programs in their appropriate sequence. The first program executed is a pre-processor program. It formats the node and line input data into a format that can be used by the rest of the code. The next program in the simulation estimates the power flow in the network. It assigns initial values to node voltages and line currents. Using the power flow equations, the voltages and current flows of the connected system are calculated.

The third executable program uses the node voltage, line current and power flow data to build the line data required by the simulation program. The simulation program will generate three output files that contain the state variables, the node voltages and the line currents. To perform the simulation the four programs need to be run in sequence since the output files of one program serve as input files for the next program.

The dynamic simulation program, developed by Professor James L. Kirtley, MIT, was originally written to simulate the dynamic response of wound field ac synchronous generators. The generators are defined as objects in the code so that multiple generators, all possibly different can be connected to the network. For this research it was necessary to generate a similar object for the permanent magnet motor. To properly interface the motor object with the other programs it was necessary to maintain the same notation as that used for the generator objects.

The simulation programs were modified to incorporate the permanent magnet motor. To incorporate the motor into the simulation program the following major modifications were made to the simulation code:

1. The state equations for the machine were modified to incorporate the permanent magnet motor model discussed in chapter three. The reference frame of the model was changed to the motor reference frame.

2. The programs were modified to delete the voltage regulator associated with each generator.

3. The ship model discussed in chapter three was incorporated into the program. This model is used to drive the windmilling shaft after the motor is disconnected from its power supply.

4. The program was modified to allow connecting a motor to the network.

5. The fault model was incorporated into the motor simulation code. The fault is introduced at a predetermined time after the simulation starts.

6. The simulation code was modified to output fault current as a function of time.

4.2 Fault simulation

The permanent magnet motor model and the parameters derived in chapters two and three were incorporated into the dynamic simulation model, Appendix E. The simplified system shown in figure 4-1 was used for the simulations. For this simulation it was assumed that a single permanent magnet motor was connected to an infinite bus. The idea of an infinite bus is not applicable to ships' power plants because loads such as propulsion motors can have ratings comparable to that of the generators.

The infinite bus assumption was made to establish the initial state of the system. For the simulation, the motor is operating at rated power and rated speed. At the time of the fault the affected motor was disconnected from the rest of the system so that there was no interaction between the motor and the rest of the system. Since interaction effects will not be addressed, the concept of an infinite bus was considered suitable for establishing the initial state of the system. This bus represented the ship's generators whose output can be adjusted to provide a specific voltage and power flow.

Using the model shown in figure 4-1, the motor was disconnected from the system some time after the initiation of the fault. In a real system this would have been accomplished with circuit breakers together with some sort of sensing system. For this simulation a fixed time of 100 milliseconds, after the initiation of the fault, was chosen for the breakers to trip and disconnect the motor from its power supply. This time was used for all simulation studies.

Once the motor was disconnected from its power supply, it was allowed to windmill as the shaft coasted down. The speed of the windmilling shaft was taken to be a function of the speed of the ship, as previously discussed in chapter three.

The fault current and power dissipated at the fault were calculated. The simulations were terminated after one second. This time was selected for the simulations since as will be shown most of the power dissipated at the fault will occur within this time. This is consistent with the results obtained in reference [19]. In addition, this short time is consistent with the assumptions made in chapter three for the ship model.

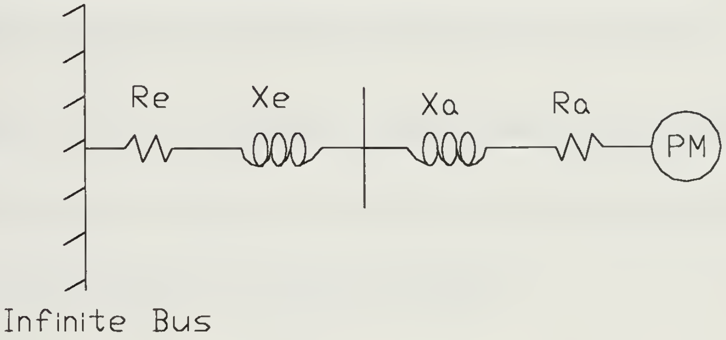

Figure 4-1. System model

The internal fault in the motor was simulated by assuming that the fault occurred somewhere in the windings between two points which have generated internal voltages and leakage inductances. A similar model was used to simulate internal faults in [19].

The motor and power supply interconnection model is shown in figure 4-2. This model consists of an internal voltage source and in series with a reactance.

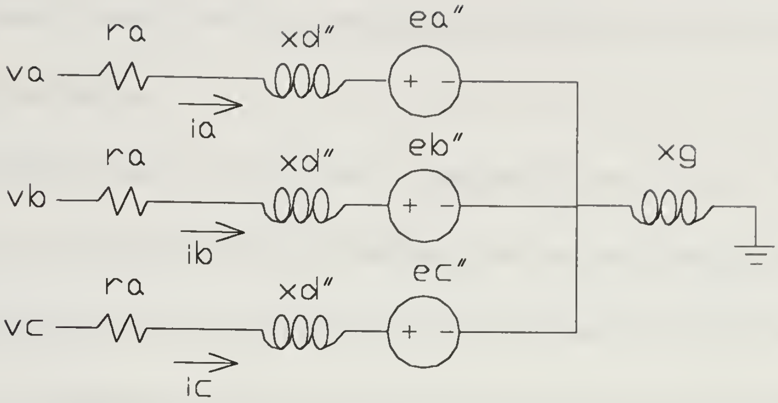

Figure 4-2. Motor and Network Interconnection Model

A derivation for this model and its representative equations are discussed in [15]. For this model it is assumed that subtransient saliency can be neglected, $x''_d = x''_q$. The internal voltages, ea'', eb'', ec'', represent voltages induced by rotor fluxes and x_g represents the impedance in the neutral of the machine and of mutual coupling between the phases. The internal voltages are calculated as follows

$$e''_a = -\frac{\omega}{\omega_o}(e''_q \sin\theta - e''_d \cos\theta) + \frac{1}{\omega_o}\cos\theta \frac{de''_q}{dt} + \frac{1}{\omega_o}\sin\theta \frac{de''_d}{dt}$$

$$e''_b = -\frac{\omega}{\omega_o}(e''_q \sin(\theta - \frac{2\pi}{3}) - e''_d \cos(\theta - \frac{2\pi}{3})) + \frac{1}{\omega_o}\cos(\theta - \frac{2\pi}{3})\frac{de''_q}{dt} + \frac{1}{\omega_o}\sin(\theta - \frac{2\pi}{3})\frac{de''_d}{dt}$$

$$e''_c = -\frac{\omega}{\omega_o}(e''_q \sin(\theta + \frac{2\pi}{3}) - e''_d \cos(\theta + \frac{2\pi}{3})) + \frac{1}{\omega_o}\cos(\theta + \frac{2\pi}{3})\frac{de''_q}{dt} + \frac{1}{\omega_o}\sin(\theta + \frac{2\pi}{3})\frac{de''_d}{dt}$$

where, θ is the shaft angle and e''_d and e''_q are the subtransient voltages defined in chapter three. A detailed derivation of this model is explained in [12] and [15].

This machine model was used in the synchronous machine simulation program. The internal fault was assumed to occur between two phase points whose voltages are proportional to the internal phase voltage. Because the fault occurred between two

phases, the voltage across the fault is proportional to the difference between the internal voltages of the two points.

The internal voltages in the machine are generated across the entire winding. Since the fault can occur anywhere in the windings there are many possible connections that can be established. For this analysis the fault was assumed to occur in the middle of the winding. The impedance seen by the fault will be proportional to the impedance of the motor winding.

The power dissipated in the fault was calculated assuming that the voltage drop across the fault was constant as discussed in chapter three.

This type of fault and simulations for a large turbogenerator are discussed in [19]. Fault currents as large as 14 per unit and peak phase currents of approximately 7 per unit were obtained in [19] for an internal phase to phase fault. These large currents can be expected to cause severe stator damage.

For an asymmetrical fault, large negative phase sequence currents are expected to flow in the stator windings and consequently in the rotor damper bars. These large currents in the damper bars could impose high thermal stresses in the damper bars.

4.3 Simulations

The following cases were run for this research and the results obtained are presented and discussed in chapter five. All the simulations were started with the motor operating at rated speed and power. A phase to phase fault was introduced, in all cases, 300 milliseconds after the simulation was initiated. The motor was disconnected from its power supply 0.1 seconds after the initiation of the fault.

For the first simulation the motor terminals remained open after the motor was disconnected from its power supply. Once the motor was disconnected it was allowed to windmill as the ship coasted down, as described in chapter three.

In the next set of simulations the motor terminals were shorted as soon as the motor was disconnected from its power supply. The motor was allowed to coast down with the terminals shorted for the remaining time of the simulation.

The same two cases discussed above were run for 25% and 75% of the winding extents.

4.4 Fault Parameters

Consistent with the fault model described in chapter three the voltage across the fault, e_a, was maintained constant at a value of 0.05 per unit. This value is based on the arc voltage values for copper electrodes given in [2] and the base voltage of Appendix A. The fault currents calculated in the model are consistent with internal fault currents as discussed in [19].

The fault current is calculated using the model presented in [2]. For a constant fault voltage the relation between fault current and voltage is given by

$$L\frac{di}{dt} + Ri + e_a = v \qquad (4\text{-}1)$$

where v is the applied voltage. Equation (4-1) represents a series R-L circuit with an applied voltage, v, and a constant voltage drop, e_a, that represents the burning voltage of the arc. For these simulations the applied voltage is proportional to the difference between two winding voltages.

If the resistance, R, is neglected compared to the inductive reactance of the circuit, and assuming e_a is constant, then equation (4-1) is solved for the fault current, i_{sc}, [2]

$$i_{sc} = -\frac{V_m}{\omega L}\cos(\omega t + \phi) + \frac{e_a}{\omega L}(\frac{\pi}{2} - \omega t)$$

where, $\phi = \cos^{-1}\frac{\pi e_a}{2V_m}$.

The instantaneous power, p_{sc}, dissipated by the fault was calculated using the product of fault current times fault voltage, or $p_{sc} = e_a i_{sc}$.

Chapter 5. Conclusions and Results

5.1 Simulation Results

The following sections describe the simulations that were run for this research. For the first set of simulations the motor was disconnected from its power supply 100 milliseconds after the fault was initiated. The terminals of the motor remained open circuited as the motor windmilled. Three cases were simulated with the fault occurring in different locations of the winding.

The same cases discussed above were run with the terminals of the motor shorted at the time the motor supply breakers opened. It was assumed that the motor terminals were shorted at the same instant that the motor was disconnected from its power supply.

5.1.1 Simulations with motor terminals open

For these simulations the motor terminals remained open after the supply breakers to the motor were opened. The internal fault occurred 300 milliseconds after the simulation started. The motor was disconnected from its power supply 100 milliseconds later and allowed to windmill. Results from this simulation are shown in figures 5-1 through 5-8. These figures correspond to a fault occurring in the middle of the windings.

The arc current reached peak levels of approximately 5 per unit, figure 5-1, while the motor was connected to the power supply. After the motor was disconnected the fault current decreased to peak values of nearly 2 per unit. As the motor slows down the

internal voltage generated inside the motor will decrease, equation (1-1). A decreasing internal voltage results in decreasing fault currents, equation (4-1).

The arc voltage is shown in figure 5-2. This voltage corresponds to a constant arc burning voltage. Using the method described in chapter four the instantaneous power dissipated at the fault was calculated, figure 5-3. For this fault the average power dissipated during the transient was determined to be approximately 0.06 per unit.

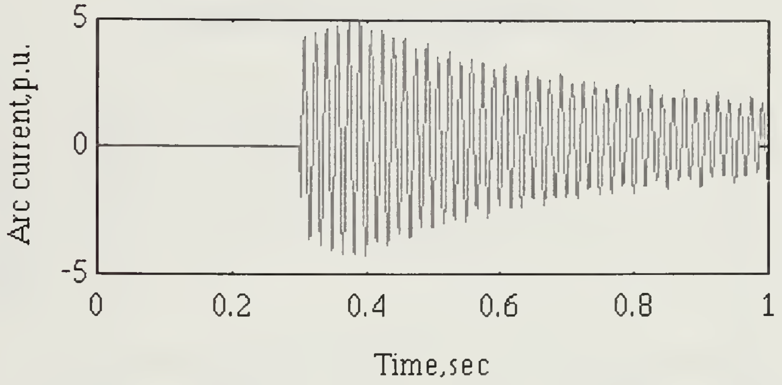

Figure 5-1. Arc current as a function of time

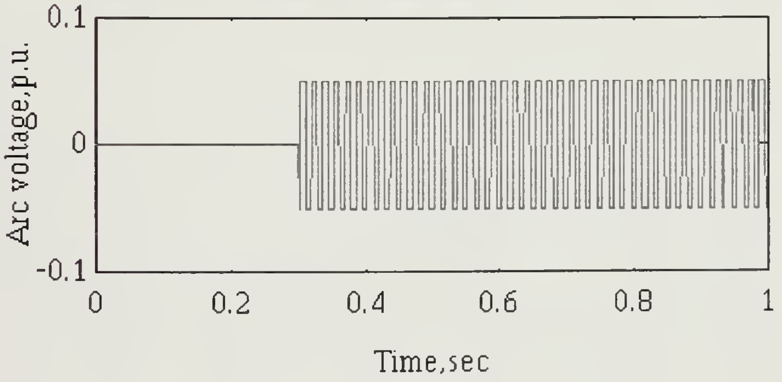

Figure 5-2. Arc voltage as a function of time

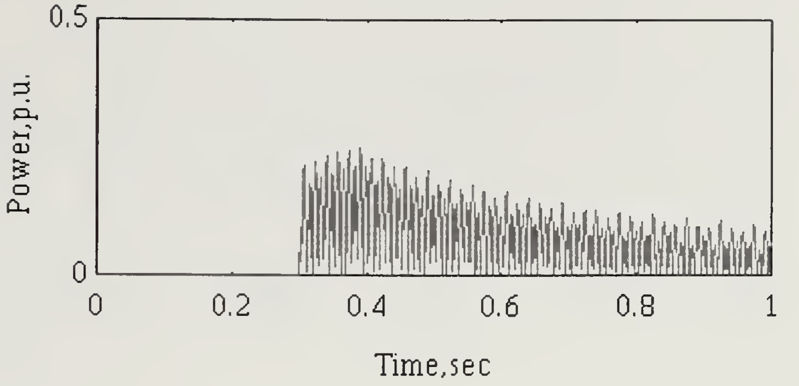

Figure 5-3. Power dissipated in arc

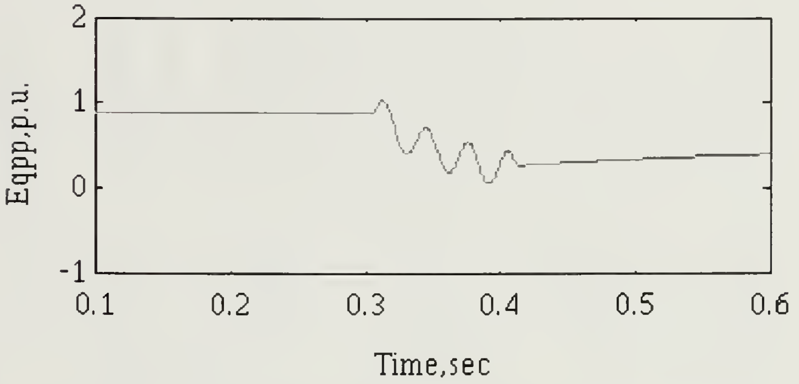

Figure 5-4. Voltage behind subtransient reactance

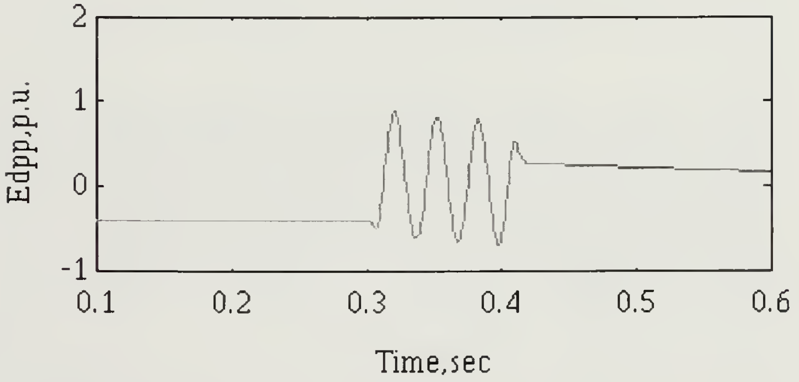

Figure 5-5. Voltage behind subtransient reactance

Figures 5-6 through 5-8 show in more detail the characteristics of the arc voltage, current and instantaneous power dissipated at the fault.

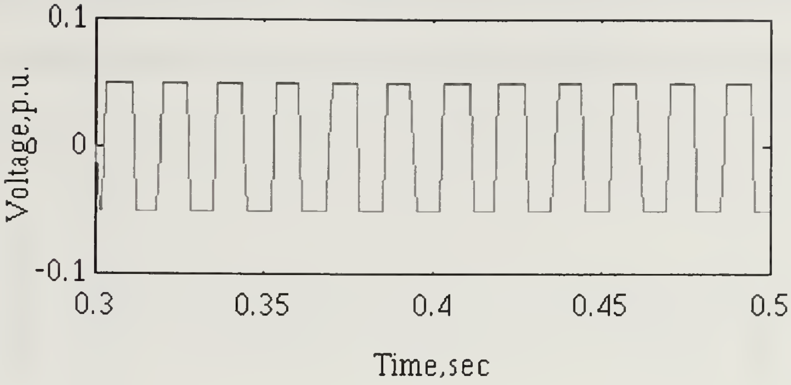

Figure 5-6. Arc voltage

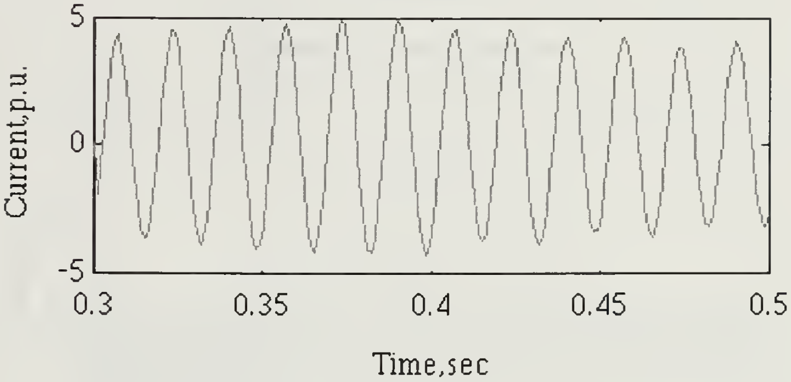

Figure 5-7. Arc current

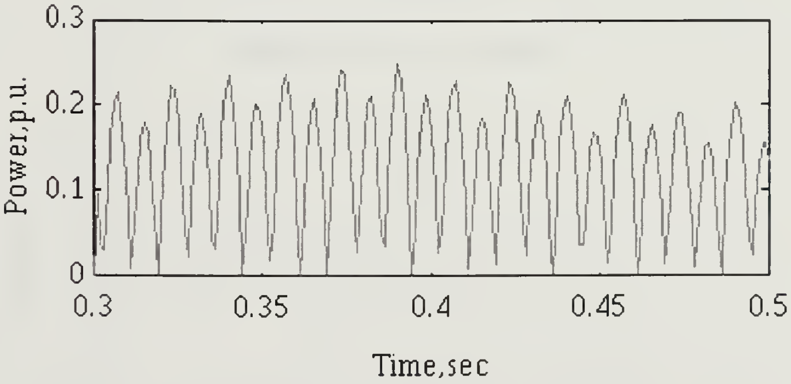

Figure 5-8. Arc power

The arc current, voltage and power, shown in figures 5-9 through 5-11, are for a location in the machine where the peak voltage is one fourth, 25% winding location, of the

internal voltage. The average power dissipated in this case over the duration of the transient, 700 milliseconds, was 0.035 per unit with a maximum current peak of 3 per unit.

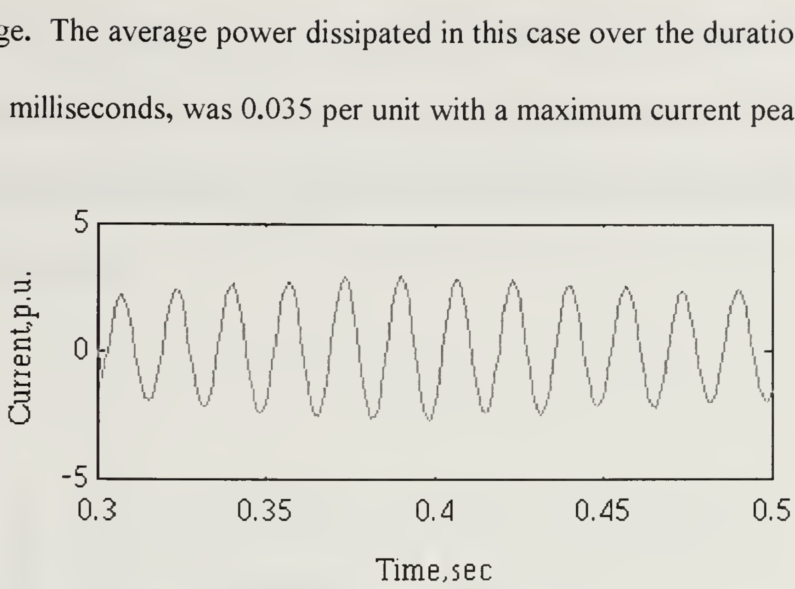

Figure 5-9. Arc current

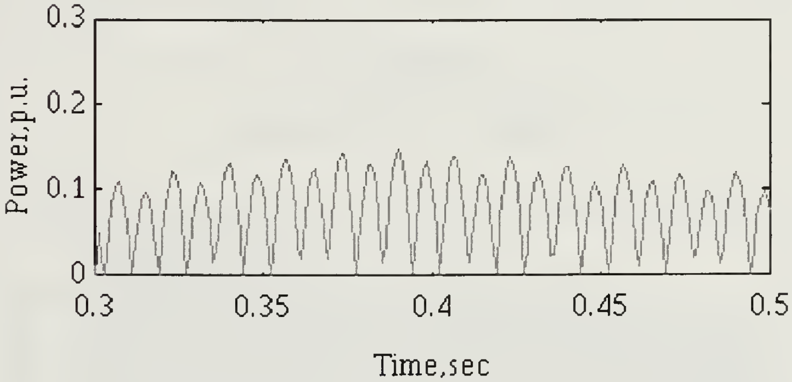

Figure 5-10. Arc power

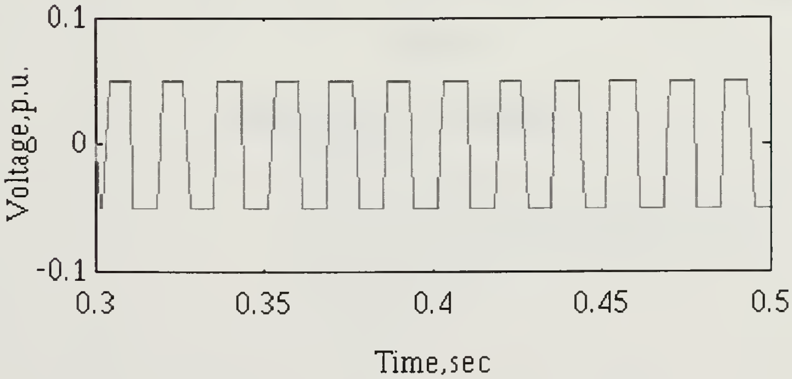

Figure 5-11. Arc voltage

Figures 5-12 through 5-14 correspond to a fault at a location where the maximum voltage is three fourths, 75% winding location, of the generated internal voltage. For this case the average power dissipated at the arc was 0.09 per unit with a maximum peak current of 6.2 per unit.

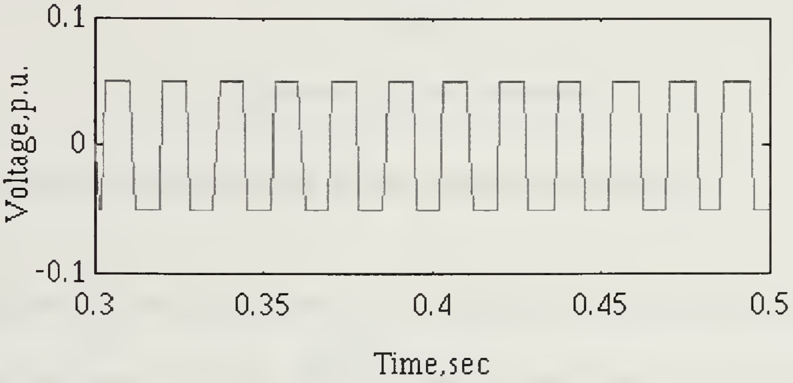

Figure 5-12. Arc voltage

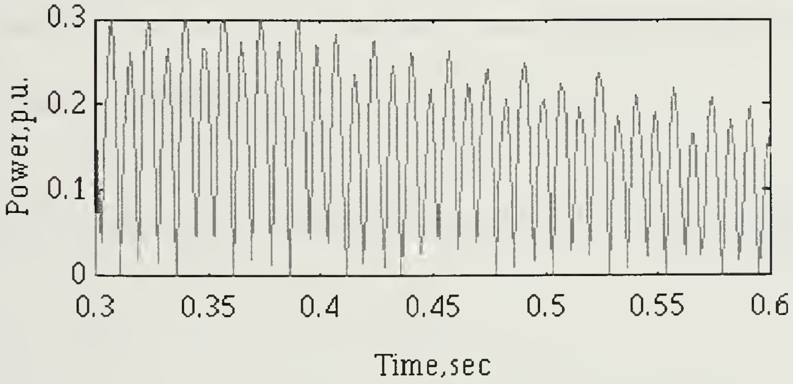

Figure 5-13. Arc power

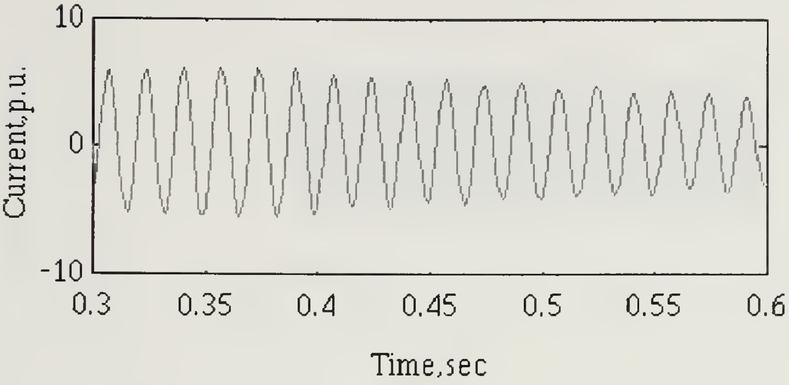

Figure 5-14. Arc current

5.1.2 Simulations with shorting of the motor terminals

The purpose of these simulations was to determine if shorting the motor terminals, after the motor has been electrically disconnected, has any effect on the power dissipation in the fault. Figures 5-15 through 5-22 show the arc currents, voltages and power dissipated at the fault. The terminals of the motor were shorted immediately after the motor supply breakers were opened.

Figures 5-15 through 5-17 correspond to fault locations near the middle of the winding.

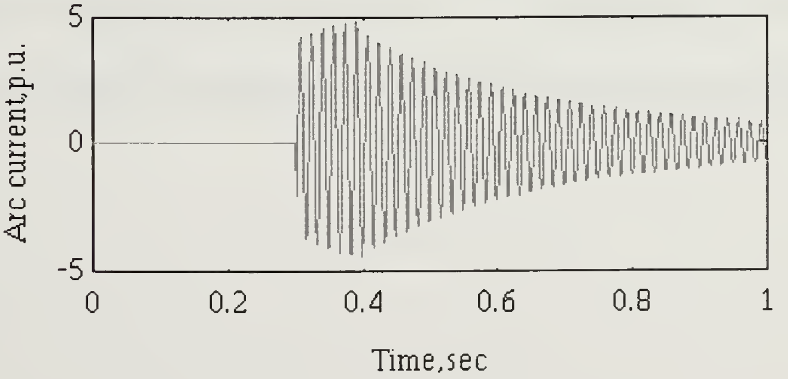

Figure 5-15. Arc current as a function of time

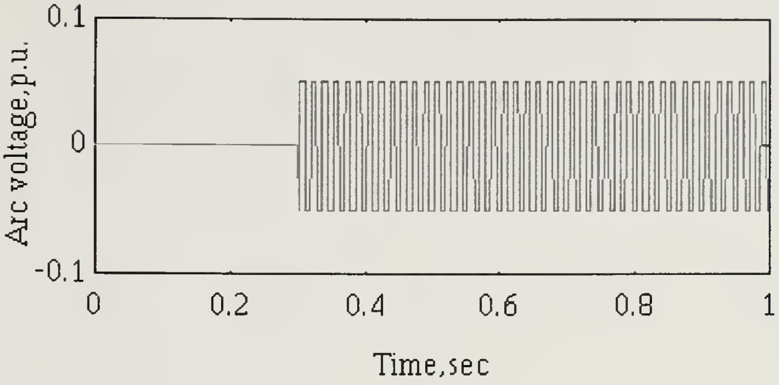

Figure 5-16. Arc voltage as a function of time

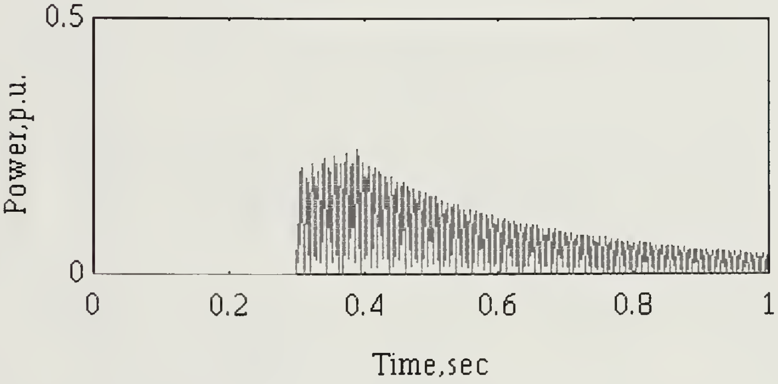

Figure 5-17. Power dissipated at fault

Shorting the terminals of the motor reduced the average power dissipated in the arc to 0.051 per unit with a peak current of 4.9 per unit. This represents an approximate 20% reduction in the power dissipated in the arc.

Figure 5-18 through 5-20 correspond to a 25% winding location. For a fault at this location the average power dissipated was 0.0295 per unit with a peak current of 3 per unit.

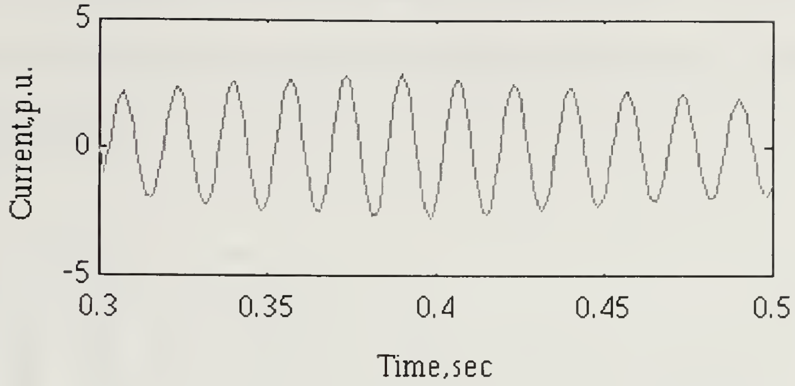

Figure 5-18. Arc current

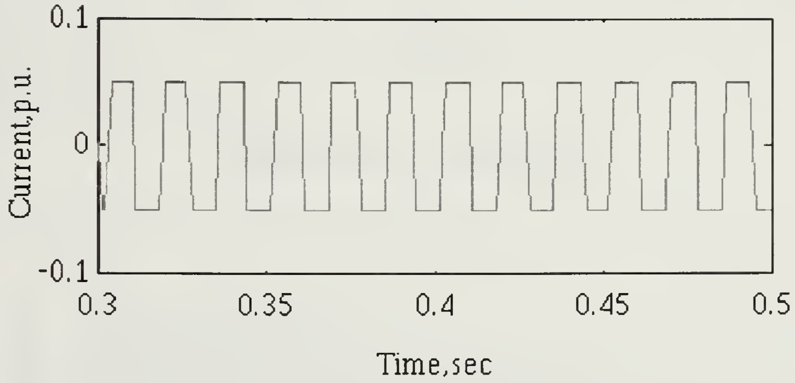

Figure 5-19. Arc voltage

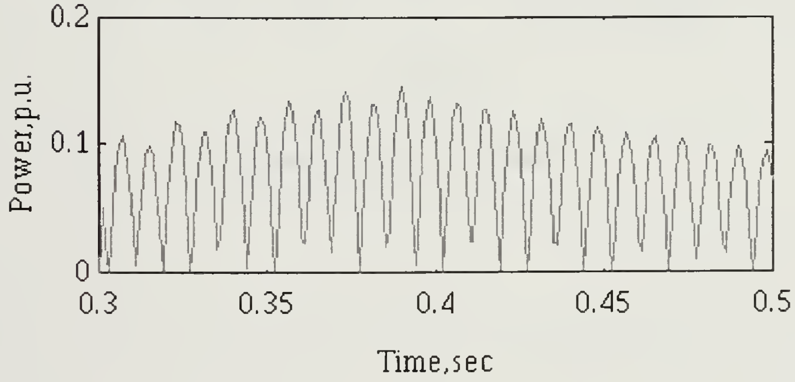

Figure 5-20. Arc power

For a 75% winding location the simulation results are shown in figures 5-21 through 5-23. In this case the average power dissipated was 0.08 per unit with a peak current of 6.2 per unit

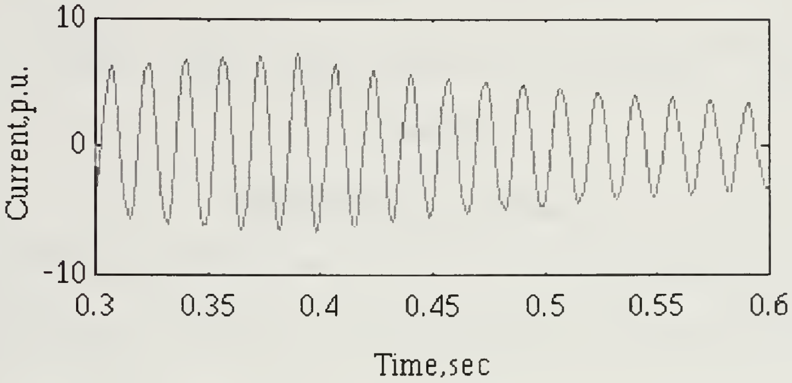

Figure 5-21. Arc current

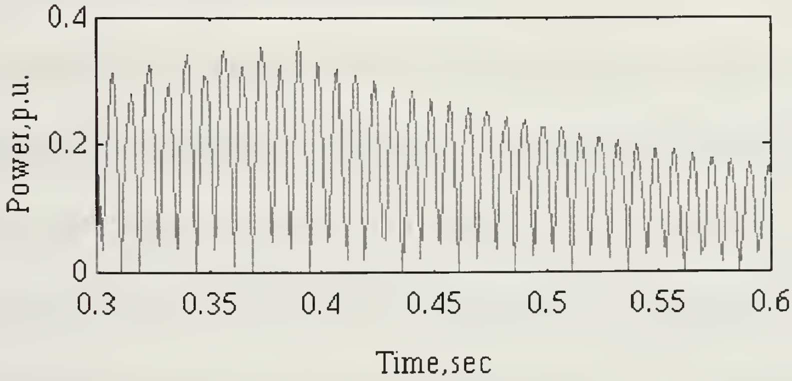

Figure 5-22. Arc power

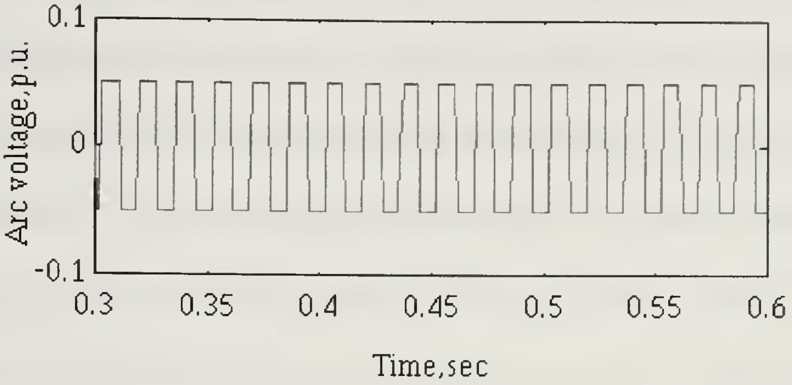

Figure 5-23. Arc Voltage

The simulations where the motor terminals were shorted showed a decrease in the average power dissipated at the fault. For all three cases, an approximate 20% reduction in dissipated power was observed.

5.2 Limitations of the simulation models

The models used in this research did not address interactions between other generators and loads during the time of the fault. In naval propulsion plants, the size of the generators can be of the same magnitude as some of the electrical loads. The effects of an internal fault on the rest of the electrical system should be considered. Such studies could be accomplished by incorporating the models developed in this research into simulation programs such as those in [4].

The only case considered in this research was that of a motor operating at rated speed and power. This case was considered since it had the capability of generating the largest fault currents. This model does not take into account how the magnitude of fault currents affects breaker tripping speeds. The effects of internal faults for motors operating at speeds lower than rated speed should be investigated.

The fault model does not consider reignition voltage of the arc. When the arc is extinguished deionization will reduce the conductivity of the column. As the voltage reverses, the conductivity that existed just before the instant the arc current goes to zero must be reestablished. If the deionization has been rapid, the required voltage to reestablish the arc will be consequently higher than the arc burning voltage. Reignition voltages can be of the order of 0.2 to 0.8 per unit, Appendix A [2]. Introducing the reignition voltage will result in a more realistic model and possibly in lower energy dissipation in the arc as the arc will not burn as frequently as in the current model.

The model used does not account for the inertia of the propeller and the shaft. This added inertia will cause the windmilling shaft to coast down at a lower rate. In this case high internal voltages will be sustained for a longer time, thus sustaining the electrical arc longer.

5.3 Suggestions for Future Research

The flexibility added by object-oriented programming makes it easier to add components to the model such as induction motors and other ship's electrical loads. This could lead to a simulation of the entire ship's electrical distribution system.

For electric drive ships, the propulsion motors can be modeled and incorporated into the simulation model.

The fault model could be improved to include the effects of arc reignition voltages. The arc reignition voltage is several times greater than the burning voltage of the arc. Inclusion of these voltages into the model will provide a more realistic model of the arc and a better estimate of the power dissipated in the arc.

The damage caused by an arc will occur in a very short period of time. The use of sensors inside the motor that can detect the formation of a plasma column should be considered. This system would disconnect the motor immediately upon the formation of the arc.

5.4 Conclusions

This research has shown that internal faults such as electrical arcs can generate large currents and dissipate large amounts of power inside a motor. Because the field flux is constant in permanent magnet machines, this type of machine will continue to generate an internal voltage while it is windmilling. This internal voltage can continue to support the fault process and could cause further damage to the machine after it is disconnected from its electrical power source.

For a large machine such as the one used for this research, average powers of approximately two to three megawatts were dissipated at the arc. Such large power dissipation in a localized spot can be expected to cause extensive damage to the motor.

Shorting the windings of the motor was shown to reduce the amount of power dissipation at the arc. However, after the windings are shorted a significant amount of power will continue to be dissipated at the arc. To stop the process the machine needs to be slowed down to where the internal voltage generated can no longer support the arcing process. Preferably the machine should be stopped.

Permanent magnet motors are being considered for ship propulsion in electric drive ships. This research has shown that an internal fault can cause significant damage to a

windmilling permanent magnet motor. Further investigation in the area of protection systems for propulsion permanent magnet motor should be considered.

References

1. Harrington, R. (ed.), *Marine Engineering*, SNAME, New York, NY, 1971.

2. Cobine, J.D., *Gaseous Conductors*, McGraw-Hill, New York, NY, 1941.

3. Crandall, S.H., Dahl, N.C., and Lardner, T.J., *An Introduction to the Mechanics of Solids*, Mc-Graw Hill, New York, NY, 1978.

4. McCoy, T.J., *Dynamic Simulation of Shipboard Electric Power Systems*, SM Thesis, Massachusetts Institute of Technology, May 1993.

5. Fitzgerald, A.E., et. al., *Electric Machinery*, Fifth Edition, McGraw-Hill, New York, NY, 1990.

6. Fink, D.G., and Beaty, H.W., *Standard Handbook for Electrical Engineers*, Thirteenth Edition, McGraw-Hill, New York, NY, 1993.

7. Bose, B.K., *Power Electronics and AC Drives*, Englewood Cliffs, NJ, 1986.

8. Leonhard, W., *Control of Electrical Drives*, Springer-Verlag, Berlin, Germany, 1985.

9. Honsinger, V.B., "Permanent Magnet Machines: Asynchronous Operation", *IEEE Transactions on Power Apparatus and Systems, Vol. PAS-99, No. 4*, pp. 1503-1509, July/August 1980.

10. *Microsoft Excel User's Guide*, Microsoft Corporation, Redmond, WA, 1992.

11. Levi, E., *Polyphase Motors*, John Wiley & Sons, New York, NY, 1984.

12. Kirtley, J.L., *Preliminary Sizing Calculations of Flux Concentrating Permanent Magnet Synchronous Machines*, Massachusetts Institute of Technology, Laboratory for Electromagnetic and Electronic Systems, April 1993.

13. Honsinger, V.B., "Performance of Polyphase Permanent Magnet Machines", *IEEE Transactions on Power Apparatus and Systems, Vol. PAS-99, No. 4*, pp. 1510-1518, July/August 1980.

14. Honsinger, V.B., "The Fields and Parameters of Interior Type AC Permanent Magnet Machines," *IEEE Transactions on Power Apparatus and Systems, Vol. PAS-101, No. 4*, pp. 867-875, April 1982

15. Kirtley, J.L., "Synchronous Machine Dynamic Models," *LEES Technical Report, TR-87-008*, Massachusetts Institute of Technology, Laboratory for Electromagnetic and Electronic Systems, 1987.

16. Gillmer, T.C., and Johnson, B., *Introduction to Naval Architecture*, Naval Institute Press, MD, 1982.

17. Lewis, E.V., (ed.), *Principles of Naval Architecture*, Volume II, SNAME, Jersey City, NJ, 1988.

18. Dewhurst, S.C., and Stark, K.T., *Programming in C++*, Prentice Hall, Englewood Cliffs, NJ, 1989.

19. Kulig, T.S., Buckley, G.W., Lambrecht, D., and Liese, M., "A New Approach to Determine Transient Generator Winding and Damper Currents in Case of Internal and External Faults and Abnormal Operation," *IEEE Transactions on Energy Conversion, Vol. 5, No. 1*, March 1990.

20. *Advanced Continuous Simulation Language User Guide/Reference Manual*, Mitchell and Gauthier, Assoc., Inc., Concord, MA, 1975.

Appendix A. Motor Design Spreadsheet

Input Parameters

No. of phases	3
L/D	0.229
Pout	40,000
Rotor speed, rpm	200
Frequency, Hz	60
Air gap size, m	0.005
Terminal Voltage, V	1000
Power factor	0.8
ks	0.9
Ja , Amps/m^2	4.00E+06
Shear, psi	7.5
Bm, T rms	0.4
Bg, T rms	0.6
Byoke, T rms	1.2
Bt, T rms	1.08
Magnet Fraction	0.5
Space Factor	0.5
q, slots/pole-phase	1
Conductors/slot	1

ω	377	r/s
p	18	pole pairs
Torque	1.42E+06	Nt-m
K	9.57E+04	A/m
R	2.12	m
B_1	0.54	T
ψ	0.64	radians
I_s	15060.0587	Amps
$\lvert 3VI \rvert$	45.18	MVA

Stator Parameters

λ_p	0.5	
Slot height	5.32	cm
N_s	15.71	turns/phase
Slot width	6.18	cm
N_{slots}	108.00	
Tooth width	6.18	cm
Back iron	5.91	cm
Eff cond length	1.44	m

Magnet Size Calculations

τ_p	37.074	cm
θ_t	0.087	radians
w_m	18.531	cm
θ_m	0.087	radians
h_m	13.903	cm
γ	0.071	
s	0.16	cm

Reactances, per unitized to Eaf

x_{ad}	0.3953	p.u.
x_{aq}	2.3128	p.u.
Zbase	0.0335	ohms
Ra	0.0005	ohms/phase
r_a	0.0145	p.u.
H	0.1227	sec

Constants, units as specified

Conductivity of Copper, S/m	5.70E+07
Density of copper, Kg/m^3	8.95E+03
Density of Steel, Kg/m^3	7.80E+03
Permeability of free space, H/m	1.26E-06
Density of Aluminum, Kg/m^3	2.70E+03

Stator Mass

Teeth	2723.35	Kg
Back iron	6227.63	Kg
Copper	2293.83	Kg
Stator Mass	11244.81	Kg
Stator Weight	11	lton

Appendix A. Motor Design Spreadsheet

Calculation of v (v= terminal voltage/Eaf)

v	e*	δ	id	eaf	vnew
2.137	1.721	0.301	0.379	0.994	2.143
2.143	1.727	0.300	0.378	1.003	2.140
2.140	1.724	0.301	0.378	0.999	2.142
2.142	1.726	0.300	0.378	1.001	2.141

	Inductance, mH	Reactance, ohms	Reactance, pu
Magnetizing	0.649	0.245	3.69
Slot leakage	0.01	0.004	0.05
Harmonic Leakage	0.016	0.006	0.09
Leakage	0.03	0.010	0.15
Laao	0.433	0.163	2.46
X_{ad}		0.012	0.366
X_{aq}		0.072	2.140
Synchronous	0.675	0.254	7.594
X_d		0.022	0.511
X_q		2.357	2.285

	Losses Mwatts	p.u.
Copper Losses	0.33	0.01
Mechanical Losses	0.000	0.000
Tooth Losses	0.00	0.0001
Core losses	0.011	0.0002
No-load losses	0.01	0.0003
Load losses	0.003	0.0001
Total losses	0.36	0.01
η	98.85%	

	Rotor Mass	
Magnets	7028.819	Kg
Poles	6573.10	Kg
Flux Barriers	10994.95	Kg
Inner Structure	3784.432	Kg
Rotor Weight	27.938	lton

Appendix B. Node and Network Pre-Processor

This program is used to format the two input files, *.in, that contain the node and line information for the simulation program. It is the first program that needs to be executed. The output of this program are two files, *.lfi which are used by the load flow calculation program, Appendix C.

The original pre-processor program in this appendix was written by Professor James L. Kirtley. The code was modified by F.R.Colberg allow connection of a synchronous motor to a network.

```
#include <stream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
 void concat(char*, char*, char*);
 char node_in_name[14];
 char line_in_name[14];
 char node_out_name[14];
 char line_out_name[14];

 concat(argv[1], ".in", node_in_name);
 concat(argv[2], ".in", line_in_name);
 concat(argv[1], ".lfi", node_out_name);
 concat(argv[2], ".lfi", line_out_name);

 filebuf f0, f1, f2, f3;
 if (f0.open(node_in_name, input) == 0)          // Bus data here
   {cout << "Can't Open " << node_in_name << "!\n";
    exit(0);
   }
 if (f1.open(line_in_name, input) == 0)          // Line data here
   {cout << "Can't Open " << line_in_name << "!\n";
    exit(0);
   }

 f2.open(node_out_name, output);                 // For processed node info
 f3.open(line_out_name, output);                // Put processed line info

 istream nodein (&f0);
 istream linein (&f1);
 ostream nodeout (&f2);
 ostream lineout (&f3);
```

```
int nlines, nnodes;
nodein >> nnodes;
linein >> nlines;

int n1[nlines], n2[nlines], line_count=0, node_count=0;
double r[nlines], x1[nlines], x0[nlines];
char s[nlines];

for (int i=0; i<nlines; i++)                // Get Basic Line Info
  {
   linein >> n1[i];
   linein >> n2[i];
   linein >> r[i];
   linein >> x1[i];
   linein >> x0[i];
   linein >> s[i];
   if (s[i] == 'c') line_count++;          // This many to output file
  }

char c;
char t[nnodes];
char gname[nnodes][10];
int node[nnodes];
double dat1[nnodes];
double dat2[nnodes];
int gen_count = 0;

for (int j=0; j<nnodes; j++)
  {
   nodein >> c;
   t[j] = c;
   switch(c)
        {
        case 'n':
          node_count++;                    // Network Node
       break;
        case 'i':                       // Voltage Source Node
          node_count++;
          nodein >> dat1[j];
          nodein >> dat2[j];
          break;
        case 'd':                       // used for datum Node
          break;                        // not in node count
        case 'v':                       // generator node
```

```
        case 'p':
          node_count++;
          dat1[j] = dat2[j] =0;
          break;
        case 'g':                        // generator or motor data
          nodein >> gname[j];
          nodein >> node[j];
          nodein >> dat1[j];
          nodein >> dat2[j];
          gen_count++;
          break;
        default:                         // unrecognizable code
          cout << "INPUT FILE ERROR: UN_RECOGNIZED NODE CODE! ";
          cout << t[j] << "\n";
          exit(0);
        }
  }
f0.close();
f1.close();

// First output file is line data for load flow

lineout << line_count << "\n";              // Top Key to line file

for (i=0; i<line_count; i++)
  if (s[i] == 'c') lineout << n1[i] << " " << n2[i] << " "
    << r[i] << " " << x1[i] << "\n";

// Second output file is node data for load flow
// First, must fix up generator buses for actual load

for (j=0; j<nnodes; j++)
  {
    if(t[j] == 'g')                     // Generator
        switch(t[node[j]])                  // Check target node
          {
          case 'v':                     // Allowed cases
            dat1[node[j]] += dat1[j];      // set voltage
              if (dat2[node[j]] == 0)
                   dat2[node[j]] = dat2[j];
            break;
          case 'p':
            dat1[node[j]] += dat1[j];      // P and Q add
            dat2[node[j]] += dat2[j];
            break;
```

```
          default:
        case 'n':
            break;
            cout << "INPUT FILE ERROR: MACHINE ASSIGNED TO WRONG
NODE!\n";
            exit(0);
          }
   }

   // Now we can actually write the output file

 nodeout << node_count << "\n";
 for (j=0; j<nnodes; j++)
   {
    switch (t[j])
        {
        case 'n':                    // Network node: special p,q
         nodeout << ".8 .6 1 0 n\n";      // p=.8,q=.6, v=1, d=0
         break;
        case 'i':                    // Voltage source
         nodeout << "0 0 ";
         nodeout << dat1[j] << " ";      // voltage
         nodeout << dat2[j] << " i\n";    // phase angle
         break;
        case 'v':                    // Generator, fixed voltage
         nodeout << dat1[j];            // total power at note
         nodeout << " 0 ";              // this number will be ignored
         nodeout << dat2[j];            // fixed voltage at node
         nodeout << " 0 v\n";           // ignored and key
         break;
        case 'p':                    // Generator, fixed p, q
         nodeout << dat1[j] << " ";      // P
         nodeout << dat2[j];            // Q
         nodeout << " 1 0 p\n";          // V to start loadflow and key
         break;
        default:                     // Only 'g' or 'd' are left
         break;
        }
   }
 f2.close();
 f3.close();
}

void concat (char *string1, char*string2, char*string)
{
```

```
 for (int j=0; j<strlen(string1); j++)
   string[j] = string1[j];
 for (int i=0; i<strlen(string2); i++)
   string[i+j] = string2[i];
}
```

Appendix C. Load Flow Program

The files *.lfi generated by the pre-processor program are used by this program to calculate node voltages, line currents and power flow. This program generates two output file *.lfo. These output files contain the load flow information for the connected system. The actual load flow calculations are performed by the program *loadflow.h*, Appendix C-1

This program was written by Professor James L. Kirtley. The program was modified by F.R.Colberg to perform load flow calculations for nodes and lines with motors connected to them.

```
#include <stream.h>
#include "loadflow.h"

main(int argc, char *argv[]
{
 void concat(char*, char*, char*);
 char node_in_name[14];
 char line_in_name[14];
 char node_out_name[14];
 char line_out_name[14];

 concat(argv[1], ".lfi", node_in_name);
 concat(argv[2], ".lfi", line_in_name);
 concat(argv[1], ".lfo", node_out_name);
 concat(argv[2], ".lfo", line_out_name);

 filebuf f0, f1, f2, f3;

 if (f0.open(node_in_name, input) == 0)          // Bus data
   {cout << "Can't Open " << node_in_name << "!\n";
    exit(0);
   }

 if (f1.open(line_in_name, input) == 0)          // Line data
   {cout << "Can't Open " << line_in_name << "!\n";
    exit(0);
   }

 f2.open(node_out_name, output);                 // Processed node information
 f3.open(line_out_name, output);                 // Processed line information

 istream nodein (&f0);
 istream linein (&f1);
```

```
  ostream nodeout (&f2);
  ostream lineout (&f3);

  const double crit=1e-7;

  cnet n (f1, f0);

  f0.close();
  f1.close();

  double e=1;
  for (int k=0; k<10000; k++)
     if((e = n.improve_voltages()) < crit) break;
  cout << "That Took " << k << " Iterations\n";

  n.report_node_voltages(f2);
  n.report_line_currents(f3);

  f2.close();
  f3.close();

}

void concat (char*string1, char*string2, char*string)
   {
   for (int j=0; j<strlen(string1); j++)
     string[j] = string1[j];
   for (int i=0; i<strlen(string2); i++)
     string[i+j] = string2[i];
   }
```

Appendix C-1. Load flow calculation program

This program solves the load flow problem for the system defined by the files *.in. This code is part of Appendix C. The original program written by Professor James L. Kirtley was modified by F.R. Colberg to add destructor functions and to delete reference to voltage regulator files.

```
#include "cline.h"
#include "cnode.h"
#include <stream.h>
#include <libc.h>
#include <stdlib.h>
#include "Complex.h"
#define Complex complex

class cnet {
 int nlines;                // number of lines
 int nnodes;               //  number of nodes in network
 cline *lptr;               // pointer to lines
 cnode *nptr;              // pointer to nodes (buses)
 int *node_incidence;
 Complex *node_admittance;

public:
 cnet (filebuf f0, filebuf f1)              // Build network from files *.lfi
  {
   int n1, n2;
   char t;
   double r, x;
   double p, q, v, d;

   istream from (&f0);                 // This file contains line information
   from >> nlines;
   lptr = new cline[nlines];           // Build the lines defined in input file
   for (int i=0; i<nlines; i++)
      {
       from >> n1;          // Entry node
       from >> n2;          // Exit node
       from >> r;           // Line resistance
       from >> x;           // Line reactance
       lptr[i].set_cline(n1, n2, r, x);
      }

   for (i=0; i<nlines; i++)
```

```
    lptr[i].rept(i);
istream fn (&f1);          // This file contains the node information
fn >> nnodes;
nptr = new cnode[nnodes];  // Build the nodes
for (int j=0; j<nnodes; j++)
    {
     fn >> p;              // Initial node real power
     fn >> q;              // Initial node reactive power
     fn >> v;              // Initial node voltage magnitude
    fn >> d;              // Initial node angle
    fn >> t;              // Node type
     nptr[j].set_cnode(p, q, v, d, t);
    }

for (j=0; j<nnodes; j++)
    nptr[j].rept(j);
node_incidence = new int[nnodes*nlines];

for (j=0; j<nnodes; j++)
    for (i=0; i<nlines; i++)
     node_incidence[i+j*nlines] = 0;

for (i=0; i<nlines; i++)      // Build node-incidence matrix
    {
     node_incidence[lptr[i].report_node_a()*nlines+i] = 1;
     node_incidence[lptr[i].report_node_b()*nlines+i] = -1;
    }

// Build the node-admittance matrix

node_admittance = new Complex[nnodes*nnodes];

for (j=0; j<nnodes; j++)                    // Allocate space for admittance matrix

    for (i=0; i<nnodes; i++)
     node_admittance[j*nnodes+i] = (0,0);

Complex *tmat = new Complex[nnodes*nlines];

for (i=0; i<nlines; i++)
    for (j=0; j<nnodes; j++)
     tmat[i*nnodes+j] = lptr[i].admit()*node_incidence[j*nlines+i];

for (j=0; j<nnodes; j++)
    for (i=0; i<nnodes; i++)
```

```
      for (int k = 0; k<nlines; k++)
       node_admittance[j*nnodes+i] +=
        node_incidence[j*nlines+k]*tmat[k*nnodes+i];

  delete [nnodes*nlines]tmat;

 }  // End of network construction step cnet()

Complex node_current(int i)          // Current at node i
 {
  Complex I;
  I = (0,0);
  for (int j = 0; j<nnodes; j++) {
   I += nptr[j].node_voltage() * node_admittance[i*nnodes+j];}
  return(I);
 }
void report_node_power()
 {
  Complex I, P;
  for (int j = 0; j<nnodes; j++)
   {
   I = (0,0);
   for (int i=0; i<nnodes; i++)
   I += nptr[i].node_voltage() * node_admittance[j*nnodes+i];
   P = nptr[j].node_voltage() * conj(I);
   cout << "Node " << j << " P + j Q = " << P << "\n";
   }
 }

double improve_voltages()            // One step in voltage improvement
 {
  Complex C, Y;
  double e = 0;          // error accumulator
  for (int i=0; i<nnodes; i++)
   {
   C = node_current(i);
   Y = node_admittance[i+nnodes*i];
   e += nptr[i].improve_voltage(C, Y)/nnodes;
   }
  return(e);
 }

void report_node_voltages(filebuf f2)       //Output node voltages
 {
  Complex V, I, P;
```

```
    ostream to(&f2);

    for (int i=0; i<nnodes; i++)
     {
     V = nptr[i].node_voltage();
     I = node_current(i);
     P = V * conj(I);
     to << abs(V) << " " << arg(V) << " "
         << real(P) << " " << imag(P) << "\n";
     }
  }
 void report_line_currents(filebuf f3)          //Output line currents
  {
    Complex c, v1, v2, I;
    int n1, n2;
    ostream to(&f3);
    for (int i=0; i<nlines; i++)
         {
          c = lptr[i].admit();
          n1 = lptr[i].report_node_a();
          n2 = lptr[i].report_node_b();
          v1 = nptr[n1].node_voltage();
          v2 = nptr[n2].node_voltage();
          I = c * (v1 - v2);
          to << abs(I) << " " << arg(I) << "\n";
         }
        }

 ~cnet();        //Destructor function
};       // End of definition of class cnet

cnet::~cnet() {
 delete [nlines]lptr;
 delete [nnodes]nptr;
 delete [nnodes*nnodes]node_admittance;
 delete [nnodes*nlines]node_incidence;
}
```

Appendix C-2. Node Voltage Calculation

This file calculates node voltages for the load flow program. It is part of Appendix C and needs to be compiled as part of it. This file was written by Professor James L. Kirtley.

```
#include "Complex.h"
#include <stream.h>
#include <stdlib.h>
#define complex Complex

class cnode {
 double p, q, v, d;          // Real, reactive power, voltage, angle
 char t;                     // Node type
public:
 void set_cnode (double pwr, double qwr, double vt, double delt, char type)
  {
   p = pwr;
   q = qwr;
   v = vt;
   d = delt;
   t = type;
  }

 Complex node_voltage()       // Report voltage at node i
  {
   return(polar(v, d));
  }

 double improve_voltage(Complex I, Complex Y) // One step in a Gauss-Seidel
  {                                           // I is current from network
   Complex cv, vo;
   double qt, e, a=1.6;
   switch(t)
      {
      case 'i':          // Infinite bus
        e = 0;
        break;
      case 'p':          // Defined p-q node
        vo = polar(v, d);
        cv = -Complex(p,-q)/(conj(vo)*Y)+ I/Y;          // Voltage correction
        vo += a*cv;
        v = abs(vo);
        d = arg(vo);
```

```
          e = abs(Complex(p,q)-vo*conj(I));
          break;
        case 'v':                          // Defined p and |v|
          vo = polar(v,d);
          qt = imag(vo*conj(I));           // Calculated value of q
          cv = Complex(p,-qt)/(conj(vo)*Y)- I/Y;
          vo += a*cv;                      // First-order correction to vo
          vo *= v/abs(vo);
          d = arg(vo);
          e = abs(p - real(vo*conj(I)));
          cout << "vo = " << vo << " I = " << I << " e = " << e << "\n";
          break;
        default:
          cout << "Incorrect Node Type Used in Input \n";
          exit(0);
        }
    return(e);
    }

  void rept(int i)
   {
    cout << " Node " << i << " Type " << t << " P= " << p << " Q= "
        << q << " V= " << v << " Delt= " << d << "\n";
   }
  Complex report_target_power()
   {
    return(Complex(p,q));
   }
};

// End of definition of class node
```

Appendix C-3. Line Admittance Calculation

This file calculates the line admittances. It is part of Appendix C the load flow program. This file was written by Professor James L. Kirtley.

```
#include "Complex.h"
typedef class complex Complex;

class cline {
 int na, nb;
 double r,x;
public:
 void set_cline (int nodea, int nodeb, double res, double react)
   {
    na = nodea;
    nb = nodeb;
    r = res;
    x = react;
   }
   Complex admit()    // Report line admittance
   {
     Complex z, y;
     z = Complex(r, x);
     y = 1.0/z;
     return(y);
   }
  int report_node_a()
   {
     return(na);
   }
  int report_node_b()
   {
     return(nb);
   }
  void repret(int i)
   {
     cout << " Line " << i << " from " << na << " to "
         << nb << " z = " << r << " + j" << x << "\n";
   }
};

// End of definition of class cline
```

Appendix D. Line Simulation Input File

This program uses the power flow information calculated in Appendix C to build the lines for the simulation program. This program will generate an output file **.net* which contains the line information for the simulation program.

Original program written by Professor James L. Kirtley. The code was modified by F.R.Colberg to delete reference to voltage regulators and to allow connection of a motor to the network.

```
#include <stream.h>
#include <stdlib.h>
#include <math.h>
#include "Complex.h"
#define Complex complex

main(int argc, char *argv[])
{
 const double a1 = 2.0943951;          // 2 pi/3

 void concat(char*, char*, char*);
 char node_in_name[14];
 char line_in_name[14];
 char node_V_name[14];
 char line_I_name[14];
 char line_out_name[14];
 char node_output_file_name[14];
 char line_output_file_name[14];
 char mot_output_file_name[14];

 concat(argv[1], ".in", node_in_name);
 concat(argv[2], ".in", line_in_name);
 concat(argv[1], ".lfo", node_V_name);
 concat(argv[2], ".lfo", line_I_name);
 concat(argv[2], ".net", line_out_name);
 concat(argv[1], ".gp", mot_output_file_name);

 filebuf f0, f1, f2, f3, f4, fg;

 if (f0.open(node_in_name, input) == 0)        // Bus data
   {cout << "Can't Open " << node_in_name << "!\n";
    exit(0);
   }
```

```
if (f1.open(line_in_name, input) == 0)           // Line data
  {cout << "Can't Open " << line_in_name << "!\n";
   exit(0);
  }

if (f2.open(node_V_name, input) == 0)            // Bus data
  {cout << "Can't Open " << node_V_name << "!\n";
   exit(0);
  }

if (f3.open(line_I_name, input) == 0)            // Line data
  {cout << "Can't Open " << line_I_name << "!\n";
   exit(0);
  }

f4.open(line_out_name, output);                  // Processed line information
fg.open(mot_output_file_name, output);

istream nodein (&f0);
istream linein (&f1);
istream nvin (&f2);
istream liin (&f3);
ostream lineout (&f4);
ostream motout (&fg);

int nlines;
linein >> nlines;

int n1[nlines], n2[nlines];
double r[nlines], x1[nlines], x0[nlines];
char s[nlines];
int lcode[nlines];

for (int i=0; i<nlines; i++)                     // Get Basic Line Info
  {
    linein >> n1[i];
    linein >> n2[i];
    linein >> r[i];
    linein >> x1[i];
    linein >> x0[i];
    linein >> s[i];
  }

 linein >> line_output_file_name;
```

```
int nnodes;
nodein >> nnodes;

char c;
char t[nnodes];
char mname[nnodes][10];
int node[nnodes];
double dat1[nnodes];
double dat2[nnodes];
int gen_count = 0;
double nvm[nnodes], nva[nnodes], p[nnodes], q[nnodes];

cout << "Reading Network Data\n";

for (int j=0; j<nnodes; j++)
  {
    nodein >> c;
    t[j] = c;
    cout << "j = " << j << " c = " << c << "\n";
    switch(c)
         {
         case 'n':
           nvin >> nvm[j];                    // Network Node
           nvin >> nva[j];
           nvin >> p[j];
           nvin >> q[j];
           cout << j << " n nvm =" << nvm[j] << " nva =" << nva[j]
             << " p =" << p[j] << " q =" << q[j] << "\n";
           break;
         case 'i':                         // Voltage Source Node
           nodein >> dat1[j];
           nodein >> dat2[j];
           nvin >> nvm[j];
           nvin >> nva[j];
           nvin >> p[j];
           nvin >> q[j];
           cout << j << " i nvm =" << nvm[j] << " nva =" << nva[j]
             << " p =" << p[j] << " q =" << q[j] << "\n";
           break;
         case 'd':                         // used for datum Node
           break;                          // not in node count
         case 'v':                         // generator node
           dat1[j] = dat2[j] =0;
           nvin >> nvm[j];
           nvin >> nva[j];
```

```
          nvin >> p[j];
          nvin >> q[j];
          cout << j << " v  nvm =" << nvm[j] << " nva =" << nva[j]
            << " p =" << p[j] << " q =" << q[j] << "\n";
          break;
        case 'p':
          dat1[j] = dat2[j] =0;
          nvin >> nvm[j];
          nvin >> nva[j];
          nvin >> p[j];
          nvin >> q[j];
          cout << j << " p  nvm =" << nvm[j] << " nva =" << nva[j]
            << " p =" << p[j] << " q =" << q[j] << "\n";
          break;
        case 'g':                              // motor data
          nodein >> mname[j];
          nodein >> node[j];
          nodein >> dat1[j];
          nodein >> dat2[j];
          gen_count++;
          break;
        default:                               // unrecognizable code
          cout << "INPUT FILE ERROR: UN_RECOGNIZED NODE CODE! ";
          cout << t[j] << "\n";
          exit(0);
          break;
        }
  }

cout << "Node Data\n";
for (j=0; j<nnodes; j++)
    cout << "Node "<<j<<"\tv= "<<nvm[j]<<"\ta= "<<nva[j]<<"\n";

int ntlines = nlines + gen_count;          // total number of lines

for (i=0; i<nlines; i++)
  linein >> lcode[i];

lineout << ntlines << "\n";                // start output

double ia, ib, ic;
int sa, sb, sc;
double lim, lia;

for (i=0; i<nlines; i++)                   // ordinary lines first
```

```
  {
   if (s[i] == 'o')                           // open line
        {
         ia = ib = ic = 0;
         sa = sb = sc = 0;
        }
   else
        {
         liin >> lim;
         liin >> lia;
         ia = lim * cos (lia);
         ib = lim * cos (lia - a1);
         ic = lim * cos (lia + a1);
         sa = sb = sc = 1;
        }
   lineout << n1[i] << " " << n2[i] << " "
        << r[i] << " " << x1[i] << " " << x0[i] << " "
        << sa << " " << sb << " " << sc << " "
        << ia << " " << ib << " " << ic << " "
        << "\n";
  }

f0.close();
f1.close();
f2.close();
f3.close();

//Build the machine connection stubs

double gx1, gx0, ra, tdopp, tqopp;
double xd, xq, xdpp, xqpp, ta, h, omz, qfract;
double P, Q;
complex I, V;

filebuf f5;
i=0;
int k;

for (j=0; j<nnodes; j++)
  {
   if (t[j] == 'g')
        {
         if (f5.open (mname[j], input) == 0)
           {
            cout << "Missing Machine File " << mname[j] << "\n";
```

```
        exit(0);
      }
    istream gin (&f5);

    gin >> xd;
   gin >> xq;
  gin >> xdpp;
  gin >> xqpp;
  gin >> gx0;
  gin >> tdopp;
  gin >> tqopp;
  gin >> ta;
  gin >> h;
  gin >> omz;

f5.close();

    gx1 = 0.5 * (xdpp + xqpp);        // positive sequence reactance
    ra = gx1 / (omz * ta);            // line resistance

     // Have to get current for the machine


    switch(t[node[j]])
       {
       case 'p':
        P = dat1[j];
        Q = dat2[j];
        break;
       case 'v':
        P = dat1[j];
        qfract=1;
        for (k=0; k<nnodes; k++)
           {
           if ((k<j) && (t[k]=='g') && (node[k] == node[j]))
            {
             qfract = dat2[j];
             break;
            }
           else if ((k>j) && (t[k] == 'g') && (node[k] == node[j]))
            qfract -= dat2[k];
          }
        Q = qfract*q[node[j]];
        break;
       default:
```

```
        cout << " Apparent Generator At Non-Generator Node!\n";
        exit(0);
       }

      I = conj(complex(P,Q)/polar(nvm[node[j]], nva[node[j]]));

      cout << "Generator Stub: P = " << P << " Q = " << Q << "\n";
      cout << "Node " << j << " To " << node[j] << " I = " << I << "\n";
      cout << "Node v = " << nvm[node[j]] << " Angle = " << nva[node[j]]
       << "\n";
      motout << nvm[node[j]] << " " << nva[node[j]] << " "
       << P << " " << Q << " " << gx1 << " " << ra << "\n";

      lim = abs(I);
      lia = arg(I);
      cout << "I = " << lim << "\tAngle = " << lia << "\tra = "
       << ra << "\tgx1 = " << gx1 << "\n";

      ia = lim * cos (lia);
      ib = lim * cos (lia - a1);
      ic = lim * cos (lia + a1);
      sa = sb = sc = 1;
      lineout << j << " " << node[j] << " "
       << ra << " " << gx1 << " " << gx0 << " "
        << sa << " " << sb << " " << sc << " "
           << ia << " " << ib << " " << ic << " "
            << "\n";
     }
  }
 lineout << line_output_file_name << "\n";
 for (i=0; i<nlines; i++)
  lineout << lcode[i] << "\n";
 for (i=0; i<gen_count; i++)
  lineout << "7\n";
f4.close();
fg.close();
}
void concat (char *string1, char*string2, char*string)
{
 for (int j=0; j<strlen(string1); j++)
  string[j] = string1[j];
 for (int i=0; i<strlen(string2); i++)
  string[i+j] = string2[i];
}
```

Appendix E. Synchronous Machine / Network Simulator

This program is the actual simulation program. It uses the output files generated by Appendices B through D and the file called *basc* to run the simulation. The output from the simulation goes to the output files designated in the *.in and base files.

Program written by Professor James L.Kirtley, modified by F.R.Colberg to incorporate the permanent magnet motor model into the simulation.

```
#include <stream.h>
#include <stdlib.h>
#include "Complex.h"
#include "mot1.h"
typedef class complex Complex;

main(int argc, char *argv[])
{
 double dt, omz;
 int np, no;
 const double a1 = 2.0943951;              // 2 pi/3

 void concat(char*, char*, char*);
 char node_in_name[14];
 char node_V_name[14];
 char line_in[14];
 char mot_in_name[14];

 concat(argv[1], ".in", node_in_name);
 concat(argv[1], ".lfo", node_V_name);
 concat(argv[1], ".gp", mot_in_name);
 concat(argv[2], ".net", line_in);

 filebuf f0, f1, f2, f3, fp;

 if (f0.open(node_in_name, input) == 0)
  {cout << "Can't Open " << argv[1] << "!\n";
   exit(0);
  }

 if (f1.open(node_V_name, input) == 0)          // Bus data here
  {cout << "Can't Open " << node_V_name << "!\n";
   exit(0);
  }
```

```
if (f2.open(line_in, input) == 0)           // Line data
  {cout << "Can't Open " << line_in << "!\n";
   exit(0);
  }

if (f3.open(argv[3], input) == 0)           // Base data
  {cout << "Can't Open " << argv[3] << "!\n";
   exit(0);
  }

if (fp.open(mot_in_name, input) == 0)           // Machine data
  {cout << "Can't Open " << mot_in_name << "!\n";
   exit(0);
  }

istream nodein (&f0);
istream nvin (&f1);
istream linein (&f2);
istream basein (&f3);
istream motin (&fp);

cout << "\n Getting Simulation Data From " << argv[3] << "\n";

basein >> omz;
basein >> dt;
basein >> np;
basein >> no;

int nevents;
basein >> nevents;
double event_time[nevents];
int event_line[nevents];
char event_type[nevents];

cout << " omz = " << omz << " dt = " << dt << " no = " << no
  << " np = " << np << "\n";

cout << " There are " << nevents << "Events:\n";

for (int ne=0; ne<nevents; ne++)
  {
   basein >> event_time[ne];
   basein >> event_line[ne];
   basein >> event_type[ne];
  }
```

```
for (ne=0; ne<nevents; ne++)
 cout << "  At time = " << event_time[ne] << " Line = "
  << event_line[ne] << " Type = " << event_type[ne] << "\n";

f3.close();

int nlines;
linein >> nlines;

int n1, n2, sa, sb, sc;
double r, x1, x0, ia, ib, ic;

line *lptr;

lptr = new line[nlines];

cout << "\n Getting Line Data From " << line_in << "\n";
cout << " There are " << nlines << " Lines\n";

for (int i=0; i<nlines; i++)                // Here we set up the lines
 {
   linein >> n1;
   linein >> n2;
   linein >> r;
   linein >> x1;
   linein >> x0;
   linein >> sa;
   linein >> sb;
   linein >> sc;
   linein >> ia;
   linein >> ib;
   linein >> ic;
   cout << "Line " << i << " Between Nodes " << n1 << " and " << n2 << "\n";
   cout << "x1= " << x1 << " x0= " << x0 << " r= " << r << "\n";
   lptr[i].setline(n1, n2, x1, x0, r, omz, sa, sb, sc, ia, ib, ic);
   lptr[i].setup();
 }

 char line_out_name[14];
 int line_out_key[nlines];

 linein >> line_out_name;
 for (i=0; i<nlines; i++)
   linein >> line_out_key[i];
```

```
f2.close();

filebuf fl;
fl.open(line_out_name, output);
ostream lineout (&fl);

cout << "Line Output Data Will Go To " << line_out_name << "\n";
cout << "Line Output Keys Are: ";
for (i=0; i<nlines; i++)
  cout << line_out_key[i] << " ";
cout << "\n";

cout << " Brief Line Summary\n";
for (i=0; i<nlines; i++)
  cout << " From " << lptr[i].report_node_a() << " To "
    << lptr[i].report_node_b() << " pointer " << &lptr[i] << "\n";


cout << "\n Getting Node Input Data From " << node_in_name << "\n";
cout << " Getting Load Flow Data From " << node_V_name << "\n";

int nnodes;
nodein >> nnodes;

cout << " There are " << nnodes << " Nodes\n";

char c;
char t[nnodes];
char mname[nnodes][10];
int node[nnodes];
double dat1[nnodes];
double dat2[nnodes];
int gen_count = 0;
int net_node_count = 0;
int v_node_count = 0;

double nvm[nnodes], nva[nnodes], p[nnodes], q[nnodes];
int ptr[nnodes];

for (int j=0; j<nnodes; j++)
  {
    nodein >> c;
    cout << " Node " << j << " c = " << c << "\n";
    switch(c)
```

```
{
case 'n':
 nvin >> nvm[j];                    // Network Node
 nvin >> nva[j];
 nvin >> p[j];
 nvin >> q[j];
 cout << "Network (type n) Node\n";
 cout << j << " n nvm =" << nvm[j] << " nva =" << nva[j]
   << " p =" << p[j] << " q =" << q[j] << "\n";
 t[j] = 'n';
 ptr[j] = net_node_count;
 net_node_count++;
 break;
case 'i':                           // Voltage Source Node
 nodein >> dat1[j];
 nodein >> dat2[j];
 nvin >> nvm[j];
 nvin >> nva[j];
 nvin >> p[j];
 nvin >> q[j];
 cout << "Voltage Source (type i) Node\n";
 cout << j << " i nvm =" << nvm[j] << " nva =" << nva[j]
   << " p =" << p[j] << " q =" << q[j] << "\n";
 t[j] = 'v';
 ptr[j] = v_node_count;
 v_node_count++;
 break;
case 'd':                           // Datum Node
 t[j] = 'v';
 ptr[j] = v_node_count;
 v_node_count++;
 dat1[j] = dat2[j] = nvm[j] = nva[j] = 0;
 cout << "Datum (type d) Node\n";
 break;                             //  Voltage node
case 'v':                           // Generator node
 dat1[j] = dat2[j] =0;
 nvin >> nvm[j];
 nvin >> nva[j];
 nvin >> p[j];
 nvin >> q[j];
 cout << "Generator (type v) Node\n";
 cout << j << " v  nvm =" << nvm[j] << " nva =" << nva[j]
   << " p =" << p[j] << " q =" << q[j] << "\n";
 t[j] = 'n';
 ptr[j] = net_node_count;
```

```
          net_node_count++;
          break;
        case 'p':
          dat1[j] = dat2[j] =0;
          break;
          nvin >> nvm[j];
          nvin >> nva[j];
          nvin >> p[j];
          nvin >> q[j];
          cout << "Generator (type p) Node\n";
          cout << j << " p  nvm =" << nvm[j] << " nva =" << nva[j]
            << " p =" << p[j] << " q =" << q[j] << "\n";
          t[j] = 'n';
          ptr[j] = net_node_count;
          net_node_count++;
          break;
        case 'g':                         // Motor data
          nodein >> mname[j];
          nodein >> node[j];
          nodein >> dat1[j];
          nodein >> dat2[j];
          t[j] = 'g';
          ptr[j] = gen_count;
          cout << "Motor\n";
          cout << "Mot File " << mname[j] << "\n";
          gen_count++;
          break;
        default:                          // Unrecognizable code
          cout << "INPUT FILE ERROR: UN-RECOGNIZED NODE CODE! ";
          cout << t[j] << "\n";
          exit(0);
          break;
        }
   }

char node_out_file_name[14];
int node_out_key[nnodes];

nodein >> node_out_file_name;

for (j=0; j<nnodes; j++)
  nodein >> node_out_key[j];

f0.close();
f1.close();
```

```
cout << " Node Voltage Data Will Go To File " << node_out_file_name << "\n";
cout << " Node Keys Are: ";
for (j=0; j<nnodes; j++)
  cout << node_out_key[j] << " ";
cout << "\n";

filebuf fn;
fn.open(node_out_file_name, output);
ostream nodeout (&fn);

cout << "Network Data Read \n";

motor* motr;                          // Pointer to motors
bus* nptr;                            // Pointer to network buses
vbus* vptr;                           // Pointer to voltage buses


motr = new motor[gen_count];
nptr = new bus[net_node_count];
vptr = new vbus[v_node_count];

cout << "Space Allocated for Motors and Buses \n";

// Setup the various nodes

filebuf fg;
int jg=0;
int jn=0;
int jv=0;

line** linebuf;
int local_line_count, local_line_no;

cout << "Ready to setup nodes\n";

  for (j=0; j<nnodes; j++)
    {
        cout << "Node " << j << " Code= " << t[j] << "\n";
        switch(t[j])
          {
          case 'g':
            if (fg.open(mname[j], input) == 0)
              {
                cout << "Can't Open " << mname[j] << "! \n";
```

```
        exit(0);
      }


    motr[jg].genset(fg);

    motr[jg].set_node(j);

    jg++;
    fg.close();
    break;
  case 'v':                    // Voltage source nodes
    cout << "Node " << j << " Voltage Source V = "
      <<  dat1[j] << " Angle = " << dat2[j] << "\n";

    vptr[jv].setvbus(dat1[j], dat2[j]);
    jv++;
    break;
  case 'n':                    // Network nodes
    cout << "--Node " << j << "\n";

    local_line_count=0;
                               //  Count connected lines
    for (i=0; i<nlines; i++)
      if ((lptr[i].report_node_a() == j) ||
            (lptr[i].report_node_b() == j))
          {
            local_line_count++;
            cout << " Line " << i << "\n";
          }

    linebuf = new line *[local_line_count];
    local_line_no=0;

    for (i=0; i<nlines; i++)       // Copy the pointers
      if ((lptr[i].report_node_a() == j) ||
            (lptr[i].report_node_b() == j))
          linebuf[local_line_no++] = &lptr[i];

    nptr[jn].setbus(j, local_line_count, linebuf);

    cout << "Node " << j << " Network Node, " << local_line_count
      << " Lines Connected \n";
    for (i=0; i<local_line_count; i++)
      cout << " From " << linebuf[i]->report_node_a()
```

```
                << " To " << linebuf[i]->report_node_b()
                  << "  pointer = " << linebuf[i] << "\n";
          jn++;

          delete linebuf;
          break;
        default:
          break;
        }
    }

// Next, set the bus pointers for each of the lines

bbus* busa;
bbus* busb;
int nn;

cout << " Summary of Bus Pointers\n";
for (j=0; j<nnodes; j++)
  switch(t[j])
    {
    case 'g':
        cout << " Node " << j << " Type g " << " ptr = " << ptr[j]
          << " pointer = " << &motr[ptr[j]] << "\n";
        break;
    case 'v':
        cout << " Node " << j << " Type v " << " ptr = " << ptr[j]
          << " pointer = " << &vptr[ptr[j]] << "\n";
        break;
    case 'n':
        cout << " Node " << j << " Type n " << " ptr = " << ptr[j]
          << " pointer = " << &nptr[ptr[j]] << "\n";
        break;
    default:
        break;
    }

for (i=0; i<nlines; i++)
  {
    nn = lptr[i].report_node_a();
    switch(t[nn])
        {
        case 'g':
          busa = &motr[ptr[nn]];
          motr[ptr[nn]].set_line(&lptr[i]);
```

```
          break;
        case 'v':
          busa = &vptr[ptr[nn]];
          break;
        case 'n':
          busa = &nptr[ptr[nn]];
          break;
        }
   nn = lptr[i].report_node_b();
   switch(t[nn])
        {
        case 'g':
          busb = &motr[ptr[nn]];
          break;
        case 'v':
          busb = &vptr[ptr[nn]];
          break;
        case 'n':
          busb = &nptr[ptr[nn]];
          break;
        }

   lptr[i].set_bus_pointers(busa, busb);
  }

// Set the initial state values for the machine(s)

double VM, VA, P, Q, X, R;
double tm[gen_count], eaf[gen_count];

for (j=0; j<gen_count; j++)
  {
   motin >> VM;
   motin >> VA;
   motin >> P;
   motin >> Q;
   motin >> X;
   motin >> R;
   cout << "\n Setting Motor " << j << " Initial Conditions\n";
   cout << " vm= " << VM << " va= " << VA << " P= " << P
        << " Q= " << Q << " X= " << X << " R= " << R << "\n";
   motr[j].set_initial(VM, VA, P, Q, X, R, 1);
   eaf[j] = motr[j].get_eaf();
   tm[j] = - motr[j].get_tm();
```

```
    cout << "Eaf = " << eaf[j] << " Torque = " << tm[j] << "\n";
  }

//Report the initial conditions

cout << "\n Final Setup Report:\n";
cout << " Motor:\n";

for (j=0; j<gen_count; j++)
  {
    motr[j].report();
    cout << "\n";
  }
cout << " Network Nodes:\n";
for (j=0; j<net_node_count; j++) nptr[j].report();
cout << "\n Voltage Source Nodes:\n";
for (j=0; j<v_node_count; j++) vptr[j].report();
cout << "\n Lines:\n";
for (i=0; i<nlines; i++) lptr[i].report();
cout << "\n";

// Start the actual time-step simulation

int n, m, eptr=0, flag=1, ip=0;
double time=0;
double te[gen_count], vbus[gen_count];

// Output first data point (the initial conditions)

  for (j=0; j<gen_count; j++)           // Machine output
    motr[j].file_output(time);

lineout << time << " ";
for (i=0; i<nlines; i++)              // Output to line file
  {
    if (line_out_key[i] & 4) lineout << lptr[i].i_a() << " ";
    if (line_out_key[i] & 2) lineout << lptr[i].i_b() << " ";
    if (line_out_key[i] & 1) lineout << lptr[i].i_c() << " ";
  }
lineout << "\n";

nodeout << time << " ";

for (j=0; j<nnodes; j++)
  switch(t[j])
```

```
    {
    case 'g':
        if (node_out_key[j] & 4) nodeout << motr[ptr[j]].a_voltage()
          << " ";
        if (node_out_key[j] & 2) nodeout << motr[ptr[j]].b_voltage()
          << " ";
        if (node_out_key[j] & 1) nodeout << motr[ptr[j]].c_voltage()
          << " ";
        break;
    case 'n':
        if (node_out_key[j] & 4) nodeout << nptr[ptr[j]].a_voltage()
          << " ";
        if (node_out_key[j] & 2) nodeout << nptr[ptr[j]].b_voltage()
          << " ";
        if (node_out_key[j] & 1) nodeout << nptr[ptr[j]].c_voltage()
          << " ";
        break;
    case 'v':
        if (node_out_key[j] & 4) nodeout << vptr[ptr[j]].a_voltage()
          << " ";
        if (node_out_key[j] & 2) nodeout << vptr[ptr[j]].b_voltage()
          << " ";
        if (node_out_key[j] & 1) nodeout << vptr[ptr[j]].c_voltage()
          << " ";
        break;
    default:
        break;
    }
nodeout << "\n";

double err, tol=1e-8;

for (n=0; n<no; n++)          // Outer loop: print loop
  {
    for (m=0; m<np; m++)
        {
         time = dt * (n * np + m);    // Time counter

         if (time>=0.4) {

        for (j=0;j<gen_count;j++) {
            tm[j]= motr[j].t_sc()*sqrt(0.4/time);
          }

         }
```

```
if (time>= event_time[eptr])    // Event monitor
 {
  switch(event_type[eptr])
      {
      case 'o':
        cout << "Opening Line " << event_line[eptr] << " at time "
          << time << "\n";
        lptr[event_line[eptr]].open_a();
        lptr[event_line[eptr]].open_b();
        lptr[event_line[eptr]].open_c();
        break;
      case 'a':
        cout << "Closing Phase A of Line " << event_line [eptr]
          << " at time " << time << "\n";
        lptr[event_line[eptr]].close_a();
        lptr[event_line[eptr]].setup();
        flag=1;
        break;
      case 'b':
        cout << "Closing Phase B of Line " << event_line [eptr]
          << " at time " << time << "\n";
        lptr[event_line[eptr]].close_b();
        lptr[event_line[eptr]].setup();
        flag=1;
        break;
      case 'c':
        cout << "Closing Phase C of Line " << event_line [eptr]
          << " at time " << time << "\n";
        lptr[event_line[eptr]].close_c();
        lptr[event_line[eptr]].setup();
        flag=1;
        break;
      case 't':
        cout << "Closing All 3 Phases of Line " << event_line [eptr]
          << " at time " << time << "\n";
        lptr[event_line[eptr]].close_a();
        lptr[event_line[eptr]].close_b();
        lptr[event_line[eptr]].close_c();
        lptr[event_line[eptr]].setup();
        flag=1;
        break;
      }
  eptr++;
  }
```

```
     for (i=0; i<nlines; i++)
       if (ip = lptr[i].current_monitor())
            {
              lptr[i].re_setup(ip);
              flag = 1;
            }

   if (flag)
     {
       for (j=0; j<net_node_count; j++)
            nptr[j].setup();
       flag = 0;
     }

if (time>=0.4) {          // Motor windmilling
     omz = 377*sqrt(0.4/time);
     }

    // Runge-Kutta step1

   for (j=0; j<gen_count; j++)
     motr[j].rk1(eaf[j], tm[j], dt, time);
   for (j=0; j<v_node_count; j++)
     vptr[j].set_v(omz * time);
   err = 1;
   do
     {err = 0;
      for (j=0; j<net_node_count; j++)
        err += nptr[j].estimate_voltage(dt);
     } while (err > tol);

   for (i=0; i<nlines; i++)
     lptr[i].rk1(dt);

     // Runge-Kutta step 2

   time = dt * (n * np + m + 0.5);    // Time

   for (j=0; j<gen_count; j++)
     motr[j].rk2(eaf[j], tm[j], dt, time);
   for (j=0; j<v_node_count; j++)
     vptr[j].set_v(omz * time);
   err = 1;
```

```
do
  {err = 0;
   for (j=0; j<net_node_count; j++)
     err += nptr[j].estimate_voltage(dt);
  } while (err > tol);
for (i=0; i<nlines; i++)
  lptr[i].rk2(dt);

  // Runge-Kutta step 3

for (j=0; j<gen_count; j++)
  motr[j].rk3(eaf[j], tm[j], dt, time);
err = 1;
do
  {err = 0;
   for (j=0; j<net_node_count; j++)
     err += nptr[j].estimate_voltage(dt);
  } while (err > tol);

for (i=0; i<nlines; i++)
  lptr[i].rk3(dt);

  // Runge-Kutta step 4

time = dt * (n * np + m + 1.0);    // time

for (j=0; j<gen_count; j++)
  motr[j].rk4(eaf[j], tm[j], dt, time);
for (j=0; j<v_node_count; j++)
  vptr[j].set_v(omz * time);
err = 1;
do
  {err = 0;
   for (j=0; j<net_node_count; j++)
     err += nptr[j].estimate_voltage(dt);
  } while (err > tol);

for (i=0; i<nlines; i++)
  lptr[i].rk4(dt);

  // Wrapup the Runge-Kutta routine:

for (j=0; j<gen_count; j++)
  motr[j].rk(time);
for (i=0; i<nlines; i++)
```

```
        lptr[i].rk();

        // Now update the exciter and avr:

      for (j=0; j<gen_count; j++)
        {
         vbus[j] = motr[j].vsense();
         te[j] = -motr[j].t_e();
        }
    }                   // This is the end of the internal loop
    for (j=0; j<gen_count; j++)           // Motor output here
      motr[j].file_output(time);

lineout << time << " ";
for (i=0; i<nlines; i++)              // Output to line file
      {
        if (line_out_key[i] & 4) lineout << lptr[i].i_a() << " ";
        if (line_out_key[i] & 2) lineout << lptr[i].i_b() << " ";
        if (line_out_key[i] & 1) lineout << lptr[i].i_c() << " ";
      }
lineout << "\n";

nodeout << time << " ";

for (j=0; j<nnodes; j++)
      switch(t[j])
        {
        case 'g':
          if (node_out_key[j] & 4) nodeout << motr[ptr[j]].a_voltage()
               << " ";
          if (node_out_key[j] & 2) nodeout << motr[ptr[j]].b_voltage()
               << " ";
          if (node_out_key[j] & 1) nodeout << motr[ptr[j]].c_voltage()
               << " ";
          break;
        case 'n':
          if (node_out_key[j] & 4) nodeout << nptr[ptr[j]].a_voltage()
               << " ";
          if (node_out_key[j] & 2) nodeout << nptr[ptr[j]].b_voltage()
               << " ";
          if (node_out_key[j] & 1) nodeout << nptr[ptr[j]].c_voltage()
               << " ";
          break;
        case 'v':
          if (node_out_key[j] & 4) nodeout << vptr[ptr[j]].a_voltage()
```

```
                  << " ";
               if (node_out_key[j] & 2) nodeout << vptr[ptr[j]].b_voltage()
                  << " ";
               if (node_out_key[j] & 1) nodeout << vptr[ptr[j]].c_voltage()
                  << " ";
               break;
             default:
               break;
             }

      nodeout << "\n";


    }  // End of the outer, or print loop.

  delete [nlines]lptr;
  delete [gen_count]motr;
  delete [net_node_count]nptr;
  delete [v_node_count]vptr;
  delete busa;
  delete busb;

  fn.close();
  fl.close();

}

void concat (char *string1, char *string2, char *string)
{
  for (int j=0; j<strlen(string1); j++)
    string[j] = string1[j];
  for (int i=0; i<strlen(string2); i++)
    string[i+j] = string2[i];
}
```

Appendix E-1. Motor Object

The following program models the permanent magnet motor. It defines the class motor that is executed as part of the simulation program. This file was written by F.R. Colberg based on the original file written for generators by Professor James L. Kirtley.

```
#include <stream.h>
#include <math.h>
#include <libc.h>
#include "line.h"
#include <stdlib.h>

class motor : public bbus{
 double xd, xq, xdpp, xqpp,         // Reactances
  tdopp, tqopp,                      // Time Constants
  xz, ta,                         // Armature quantities
  h, omz, thz,                  // Miscellaneous quantities
  eqpp, edpp, om, delt,            // State Variables
  kqpp1, kdpp1, kdelt1, kom1,       // R-K derivatives
  kqpp2, kdpp2, kdelt2, kom2,       // R-K derivatives
  kqpp3, kdpp3, kdelt3, kom3,       // R-K derivatives
  kqpp4, kdpp4, kdelt4, kom4,       // R-K derivatives
  ea, eb, ec,                  // Output Voltages
  dd1, dd2, dd3, dd4,               // intermediate derivatives
  dq1, dq2, dq3, dq4,
  tmr, eafr,tm,                 // required torque and eaf
  iaz, ibz, icz,               // initial currents
  te1, te2, te3, te4, te;          // electrical torque
  int node;                    // node number of internal bus
  line* lptr;                  // line connecting machine to net
  char ofname[10];                 // file name for output
  filebuf f1;

 public:
  void genset(filebuf f0)           // Motor reads input from file
   {
    istream from (&f0);
    from >> xd;
    from >> xq;
    from >> xdpp;
    from >> xqpp;
    from >> xz;
    from >> tdopp;
    from >> tqopp;
```

```
    from >> ta;
    from >> h;
    from >> omz;
    from >> ofname;
    f1.open(ofname, output);
   }

void set_line (line* lineptr)
  {
   lptr = lineptr;
  }

void set_node (int node_number)
  {
   node = node_number;
  }

void file_output(double time)
  {
   ostream to (&f1);
   to << time << " " << eqpp << " " << edpp << " "
        << " " << delt << " " << om << "\n";
  }

void report()
  {
   cout << " Motor\n"
        << " xd = " << xd
        << " xq = " << xq
        << " xdpp = " << xdpp
        << " xqpp = " << xqpp << "\n"
        << " tdopp = " << tdopp
        << " tqopp = " << tqopp
        << " h = " << h
        << "\n"
        << " eqpp = " << eqpp
        << " edpp = " << edpp
        << " om = " << om
        << " delt = " << delt << "\n"
        << " node = " << node
        << " conn to bus " << lptr->report_node_b() << "\n";
  }

void set_initial (double vl, double thl, double pl, double ql,
                  double xl, double rl, int long_out = 0)
```

```
                                    // set initial conditions from
                                    // external bus. xl and rl are
                                    // branch inductance (including
                                    // subtransient reactance )
{
 double psi = atan2 (ql, pl);      // power factor angle at ext bus
 double il = sqrt(pl*pl+ql*ql)/vl;  // load current
 double xt = xq + xl - xdpp;        // total inductance to ext bus
 double vr = vl - rl * il * cos (psi) + xt * il * sin (psi);
 double vi = xt * il * cos (psi) + rl * il * sin (psi);
 double vs = sqrt (vr*vr + vi*vi);
 double ths = thl - atan2 (vi, vr);
 double th, ca, cb, cc, sa, sb, sc;
 double s = 2.0943951;      // 2 pi/3

 thz = thl - 1.5707963;
 delt = ths - thl;

 double id = il * sin (delt + psi);        // direct and quadrature axis currents
 double iq = il * cos (delt + psi);
 eafr = vs + id * (xd - xq);
 om = omz;

 eqpp = eafr + (xd - xdpp) * id;
 edpp = -(xq - xqpp) * iq;

 tmr = (eqpp*iq + edpp*id + (xdpp - xqpp) * id * iq);

 iaz = il * cos (thl + psi);
 ibz = il * cos (thl + psi - 2.0943951);
 icz = il * cos (thl + psi + 2.0943951);

 th = thz + delt;
 ca = cos (th); cb = cos (th - s); cc = cos (th + s);
 sa = sin (th); sb = sin (th - s); sc = sin (th + s);

 ea = - (om/omz)*(eqpp*sa-edpp*ca);
 eb = - (om/omz)*(eqpp*sb-edpp*cb);
 ec = - (om/omz)*(eqpp*sc-edpp*cc);

 set_va(ea); set_vb(eb); set_vc(ec);

 if (long_out)
   {
   cout << form("mot::set_initial() Node = %d\n", node);
```

```
    cout << form("vl = %g  thl = %g  pl = %g  ql = %g\n", vl, thl, pl, ql);
    cout << form("xl = %g  rl = %g   psi = %g  il = %g\n", xl, rl, psi, il);
    cout << form("xt = %g  vr = %g  vi = %g  vs = %g\n", xt, vr, vi, vs);
    cout << form("ths = %g  thz = %g  delt = %g  id = %g\n",ths,thz,delt, id);
    cout << form("iq = %g  eafr = %g  tmr = %g  om = %g\n", iq,eafr, tmr, om);
    cout << form("eqpp = %g  edpp = %g\n\n", eqpp, edpp);
   }

}

  double get_tm() {return (tmr);}
  double get_eaf() {return (eafr);}
  double get_ia() {return (iaz);}
  double get_ib() {return (ibz);}
  double get_ic() {return (icz);}
  double get_eqpp() {return (eqpp);}
  double get_edpp() {return (edpp);}
  double t_e() {return (te);}
  double t_e1() {return (te1);}
  double t_e2() {return (te2);}
  double t_e3() {return (te3);}
  double t_e4() {return (te4);}

  double t_sc() {              //Calculates input torque to the
    tm = get_tm();             //motor with the shaft free wheeling
    return(tm);
   }

  double eaf_sc() {
   double eaf = get_eaf();
    return (eaf);
  }

  void disp_params() {cout << " xd = " << xd
                     << " xq = " << xq
                     << "\n"
                     << " xdpp = " << xdpp
                     << " xqpp = " << xqpp << "\n"
                     << " tdopp = " << tdopp
                     << " tqopp = " << tqopp << "\n"
                     << " h = " << h
                     << " omz = " << omz << " thz =" << thz <<"\n";}
  void disp_state()  {cout << "eqpp = " << eqpp
                     << "\tedpp = " << edpp << "\n"
                     << " om = " << om << "\tdelta = " << delt << "\n"; }
```

```
 double e_a() {return(ea);}
 double e_b() {return(eb);}
 double e_c() {return(ec);}
 double omega() {return(om);}
 double delta() {return(delt);}

 void rk1 (double eaf, double tm,
            double dt, double t, int long_out = 0)
{
 double ia, ib, ic;
 double th, id, iq, deqpp, dedpp, ddelt, dom,
       ca, cb, cc, sa, sb, sc;
 double s = 2.0943951;      // 2 pi / 3

  // First, Park's Transform Currents

 ia = -lptr->i_ap();
 ib = -lptr->i_bp();
 ic = -lptr->i_cp();

 if (long_out)
   {
    cout << form("rk1:\n");
    cout << form("node %d\n",node);
    cout << form("ia = %g  ib = %g  ic = %g\n", ia, ib, ic);
    cout << form("eaf = %g  tm = %g  dt = %g  t = %g\n", eaf,tm,dt,t);
   }

 th = thz + omz * t + delt;
 ca = cos (th); cb = cos (th - s); cc = cos (th + s);
 sa = sin (th); sb = sin (th - s); sc = sin (th + s);

 id = .6666666667 * ( ia * ca + ib * cb + ic * cc );
 iq = .6666666667 * ( -ia * sa - ib * sb - ic * sc);

 if (long_out)
   {
    cout << form("th = %g  ca = %g  sa = %g\n", th, ca, sa);
    cout << form("id = %g  iq = %g\n", id, iq);
   }

  // Now do the time step

 deqpp = - eqpp/tdopp + eaf/tdopp + (xd - xdpp) * id / tdopp;
```

```
dedpp = - edpp/tqopp - (xq - xqpp) * iq / tqopp;
ddelt = om - omz;
if (t>=0.4) {
 tel = 0.0;
}
else
{tel = eqpp*iq + edpp*id +(xdpp - xqpp) * id * iq;}
dom = (omz/(2.0*h)) * (tel + tm);

if (long_out)
  {
    cout << form("deqpp = %g  dedpp = %g\n", deqpp, dedpp);
    cout << form("ddelt = %g  dom = %g\n", ddelt, dom);
  }

  // Now the R-K coefficients are:

kqpp1 = dt * deqpp;
kdpp1 = dt * dedpp;
kdelt1 = dt * ddelt;
kom1 = dt * dom;

 // save for final step

dq1 = deqpp;
dd1 = dedpp;

if (long_out)
  {
    cout << form("kqpp = %g  kdpp = %g\n", kqpp1, kdpp1);
    cout << form("kdelt = %g  kom = %g\n", kdelt1, kom1);
  }

ea = - (om/omz)*((eqpp+.5*kqpp1)*sa-(edpp+.5*kdpp1)*ca)
     + (1.0/omz)*(deqpp*ca+dedpp*sa);
eb = - (om/omz)*((eqpp+.5*kqpp1)*sb-(edpp+.5*kdpp1)*cb)
     + (1.0/omz)*(deqpp*cb+dedpp*sb);
ec = - (om/omz)*((eqpp+.5*kqpp1)*sc-(edpp+.5*kdpp1)*cc)
     + (1.0/omz)*(deqpp*cc+dedpp*sc);

set_va(ea);
set_vb(eb);
set_vc(ec);
if (t<0.3) {
  set_varc(0.5*((ea-eb)));}
```

```
  if (long_out)
    {
      cout << form("ea = %g  eb = %g  ec = %g\n\n", ea, eb, ec);
    }

}

  void rk2 (double eaf, double tm,
                double dt, double t, int long_out = 0)
{
  double th, id, iq, deqpp, dedpp, ddelt, dom,
         ca, cb, cc, sa, sb, sc;
  double s = 2.0943951;       // 2 pi / 3
  double eqppt, edppt, deltt, omt;   // temporaries for states
  double ia, ib, ic;

    // assign state temporaries:

  eqppt = eqpp + .5 * kqpp1;
  edppt = edpp + .5 * kdpp1;
  omt = om + .5 * kom1;
  deltt = delt + .5 * kdelt1;

    // First, Park's Transform Currents

  ia = -lptr->i_ap();
  ib = -lptr->i_bp();
  ic = -lptr->i_cp();

  if (long_out)
    {
      cout << form("rk2:\n");
      cout << form("node %d\n", node);
      cout << form("ia = %g  ib = %g  ic = %g\n", ia, ib, ic);
      cout << form("eaf = %g  tm = %g  dt = %g  t = %g\n", eaf,tm,dt,t);
    }

  th = thz + omz * t + delt;
  ca = cos (th); cb = cos (th - s); cc = cos (th + s);
  sa = sin (th); sb = sin (th - s); sc = sin (th + s);

  id = .6666666667 * ( ia * ca + ib * cb + ic * cc );
  iq = .6666666667 * ( -ia * sa - ib * sb - ic * sc);
```

```
if (long_out)
  {
    cout << form("th = %g  ca = %g  sa = %g\n", th, ca, sa);
    cout << form("id = %g  iq = %g\n", id,iq);
  }

 // Now do the time step

deqpp = - eqppt/tdopp + eaf/tdopp + (xd - xdpp) * id / tdopp;
dedpp = - edppt/tqopp - (xq - xqpp) * iq / tqopp;
ddelt = omt - omz;
if (t>=0.4) {
 te2=0.0;}
else
{te2 = eqppt*iq + edppt*id +(xdpp - xqpp) * id * iq;}
dom = (omz/(2.0*h)) * (te2 + tm);

if (long_out)
  {
    cout << form("deqpp = %g  dedpp = %g\n", deqpp, dedpp);
    cout << form("ddelt = %g  dom = %g\n", ddelt,dom);
  }

 // Now the R-K coefficients are:

kqpp2 = dt * deqpp;
kdpp2 = dt * dedpp;
kdelt2 = dt * ddelt;
kom2 = dt * dom;

 // save for final step

dq2 = deqpp;
dd2 = dedpp;

if (long_out)
  {
    cout << form("kqpp = %g  kdpp = %g\n", kqpp1, kdpp1);
    cout << form("kdelt = %g  kom = %g\n", kdelt1, kom1);
  }

ea = - (om/omz)*((eqpp+.5*kqpp2)*sa-(edpp+.5*kdpp2)*ca)
    + (1.0/omz)*(deqpp*ca+dedpp*sa);
eb = - (om/omz)*((eqpp+.5*kqpp2)*sb-(edpp+.5*kdpp2)*cb)
    + (1.0/omz)*(deqpp*cb+dedpp*sb);
```

```
 ec = - (om/omz)*((eqpp+.5*kqpp2)*sc-(edpp+.5*kdpp2)*cc)
     + (1.0/omz)*(deqpp*cc+dedpp*sc);

 set_va(ea);
 set_vb(eb);
 set_vc(ec);

if (t<0.3) {
   set_varc(0.5*((ea-eb));}

 if (long_out)
   {
    cout << form("ea = %g  eb = %g  ec = %g\n\n", ea, eb, ec);
   }

}

 void rk3 (double eaf, double tm,
           double dt, double t, int long_out = 0)
{
 double th, id, iq, deqpp, dedpp, ddelt, dom,
        ca, cb, cc, sa, sb, sc;
 double s = 2.0943951;       // 2 pi / 3
 double eqppt, edppt, deltt, omt; // temporaries for states
 double ia, ib, ic;

   // assign state temporaries:

 eqppt = eqpp + .5 * kqpp2;
 edppt = edpp + .5 * kdpp2;
 omt = om + .5 * kom2;
 deltt = delt + .5 * kdelt2;

   // First, Park's Transform Currents

 ia = -lptr->i_ap();
 ib = -lptr->i_bp();
 ic = -lptr->i_cp();

 if (long_out)
   {
    cout << form("rk3:\n");
    cout << form("node %d\n", node);
    cout << form("ia = %g  ib = %g  ic = %g\n", ia, ib, ic);
    cout << form("eaf = %g  tm = %g  dt = %g  t = %g\n", eaf,tm,dt,t);
```

```
  }

th = thz + omz * t + delt;
ca = cos (th); cb = cos (th - s); cc = cos (th + s);
sa = sin (th); sb = sin (th - s); sc = sin (th + s);

id = .6666666667 * ( ia * ca + ib * cb + ic * cc );
iq = .6666666667 * ( -ia * sa - ib * sb - ic * sc);

if (long_out)
  {
    cout << form("th = %g  ca = %g  sa = %g\n", th, ca, sa);
    cout << form("id = %g  iq = %g\n", id,iq);
  }

  // Now do the time step

deqpp = - eqppt/tdopp + eaf/tdopp + (xd - xdpp) * id / tdopp;
dedpp = - edppt/tqopp - (xq - xqpp) * iq / tqopp;
ddelt = omt - omz;
 if (t>=0.4) {
  te2=0.0;}
else
 {te3 = eqppt*iq + edppt*id +(xdpp - xqpp) * id * iq;}
dom = (omz/(2.0*h)) * (te3 + tm);

 // save for final step

dq3 = deqpp;
dd3 = dedpp;

if (long_out)
  {
    cout << form("deqpp = %g  dedpp = %g\n", deqpp, dedpp);
    cout << form("ddelt = %g  dom = %g\n", ddelt,dom);
  }

  // Now the R-K coefficients are:

kqpp3 = dt * deqpp;
kdpp3 = dt * dedpp;
kdelt3 = dt * ddelt;
kom3 = dt * dom;

if (long_out)
```

```
  {
   cout << form("kqpp = %g  kdpp = %g\n", kqpp1, kdpp1);
   cout << form("kdelt = %g  kom = %g\n", kdelt1, kom1);
  }

  // And now for the voltages:


 ea = - (om/omz)*((eqpp+kqpp3)*sa-(edpp+kdpp3)*ca)
     + (1.0/omz)*(deqpp*ca+dedpp*sa);
 eb = - (om/omz)*((eqpp+kqpp3)*sb-(edpp+kdpp3)*cb)
     + (1.0/omz)*(deqpp*cb+dedpp*sb);
 ec = - (om/omz)*((eqpp+kqpp3)*sc-(edpp+kdpp3)*cc)
     + (1.0/omz)*(deqpp*cc+dedpp*sc);

  set_va(ea);
  set_vb(eb);
  set_vc(ec);

 if (t<0.3) {
  set_varc(0.5*((ea-eb)));}

 if (long_out)
  {
   cout << form("ea = %g  eb = %g  ec = %g\n\n", ea, eb, ec);
  }

}

 void rk4 (double eaf, double tm,
           double dt, double t, int long_out = 0)
{
 double th, id, iq, deqpp, dedpp, ddelt, dom,
       ca, cb, cc, sa, sb, sc;
 double s = 2.0943951;      // 2 pi / 3
 double eqppt, edppt, deltt, omt; // temporaries for states
 double ia, ib, ic;

  // assign state temporaries:

 eqppt = eqpp + kqpp3;
 edppt = edpp + kdpp3;
 omt = om + kom3;
 deltt = delt + kdelt3;
```

```
 // First, Park's Transform Currents

ia = -lptr->i_ap();
ib = -lptr->i_bp();
ic = -lptr->i_cp();

if (long_out)
  {
   cout << form("rk4:\n");
   cout << form("node %d\n", node);
   cout << form("ia = %g  ib = %g  ic = %g\n", ia, ib, ic);
   cout << form("eaf = %g  tm = %g  dt = %g  t = %g\n", eaf,tm,dt,t);
  }

th = thz + omz * t + delt;
ca = cos (th); cb = cos (th - s); cc = cos (th + s);
sa = sin (th); sb = sin (th - s); sc = sin (th + s);

id = .6666666667 * ( ia * ca + ib * cb + ic * cc );
iq = .6666666667 * ( -ia * sa - ib * sb - ic * sc);

if (long_out)
  {  cout << form("th = %g  ca = %g  sa = %g\n", th, ca, sa);
     cout << form("id = %g  iq = %g\n", id,iq);
  }

 // Now do the time step

deqpp = - eqppt/tdopp + eaf/tdopp + (xd - xdpp) * id / tdopp;
dedpp = - edppt/tqopp - (xq - xqpp) * iq / tqopp;
ddelt = omt - omz;
 if (t>=0.4) {
 te2=0.0;}
else
{te4 = eqppt*iq + edppt*id +(xdpp - xqpp) * id * iq;}
dom = (omz/(2.0*h)) * (te4 + tm);

if (long_out)
  {
   cout << form("deqpp = %g  dedpp = %g\n", deqpp, dedpp);
   cout << form("ddelt = %g  dom = %g\n", ddelt,dom);
  }

  //  The R-K coefficients are:
```

```
 kqpp4 = dt * deqpp;
 kdpp4 = dt * dedpp;
 kdelt4 = dt * ddelt;
 kom4 = dt * dom;

 if (long_out)
   {
     cout << form("kqpp = %g  kdpp = %g\n", kqpp1, kdpp1);
     cout << form("kdelt = %g  kom = %g\n", kdelt1, kom1);
   }

  // save for final step

 dq4 = deqpp;
 dd4 = dedpp;

 ea = - (om/omz)*(eqpp*sa-edpp*ca)+(1.0/omz)*(deqpp*ca+dedpp*sa);
 eb = - (om/omz)*(eqpp*sb-edpp*cb)+(1.0/omz)*(deqpp*cb+dedpp*sb);
 ec = - (om/omz)*(eqpp*sc-edpp*cc)+(1.0/omz)*(deqpp*cc+dedpp*sc);

  set_va(ea);
  set_vb(eb);
  set_vc(ec);

if (t<0.3) {
   set_varc(0.5*((ea-eb)));}

 if (long_out)
   {
     cout << form("ea = %g  eb = %g  ec = %g\n\n", ea, eb, ec);
   }

}

// End the Runge-Kutta routine

 void rk(double t, int long_out = 0)
{
  double th, ca, cb, cc, sa, sb, sc;
  double ia, ib, ic, id, iq;
  double s = 2.0943951;       // 2 pi / 3

  ia = -lptr->i_ap();
  ib = -lptr->i_bp();
  ic = -lptr->i_cp();
```

```
th = thz + omz * t + delt;
ca = cos (th); cb = cos (th - s); cc = cos (th + s);
sa = sin (th); sb = sin (th - s); sc = sin (th + s);

id = .6666666667 * ( ia * ca + ib * cb + ic * cc );
iq = .6666666667 * ( -ia * sa - ib * sb - ic * sc);

eqpp += (kqpp1 + 2*kqpp2 + 2*kqpp3 + kqpp4)/6.0;
edpp += (kdpp1 + 2*kdpp2 + 2*kdpp3 + kdpp4)/6.0;
delt += (kdelt1 + 2*kdelt2 + 2*kdelt3 + kdelt4)/6.0;
om += (kom1 + 2*kom2 + 2*kom3 + kom4)/6.0;

double deqpp = (dq1 + 2*dq2 + 2*dq3 + dq4)/6;
double dedpp = (dd1 + 2*dd2 + 2*dd3 + dd4)/6;

ea = - (om/omz)*(eqpp*sa-edpp*ca)+(1.0/omz)*(deqpp*ca+dedpp*sa);
eb = - (om/omz)*(eqpp*sb-edpp*cb)+(1.0/omz)*(deqpp*cb+dedpp*sb);
ec = - (om/omz)*(eqpp*sc-edpp*cc)+(1.0/omz)*(deqpp*cc+dedpp*sc);

 set_va(ea);
 set_vb(eb);
 set_vc(ec);

 if (t<0.3) {
  set_varc(0.5*((ea-eb)));}

te = (te1 + 2.0 * te2 + 2.0 * te3 + te4)/6.0;

if (long_out)
  {
   cout << form("rk:\n");
   cout << form("eqpp = %g  edpp = %g\n", eqpp, edpp);
   cout << form("delt = %g  om = %g  \n\n", delt, om);
  }

filebuf fr;
char int_volt[15] ="int.dat";
double i_sc, e_sc, p_sc;
fr.open(int_volt, append);
ostream voltout(&fr);
voltout<<t<<" ";
if ((t>0.3)&&((0.5*(ea-eb)>0.05)||(0.5*(ea-eb)<-0.05))) {
  i_sc = -0.5*(ea-eb)/xdpp*cos(om*t + acos(0.07854/(varc)))
          + 0.05/xdpp(1.57079-om*t);}      //Calculation of short circuit current
```

```
   else {
     i_sc = 0.0;}
 if (i_sc>0.0)
   {e_sc=0.05;}
 else
  if ( i_sc< 0.0)
    {e_sc=-0.05;}
  else
 {e_sc= 0.0}

 p_sc=e_sc*i_sc;

  voltout<<e_sc<<" "<<i_sc<<" ";
   voltout<<p_sc<<" ";
   voltout<<"\n";
 }
 };
//End of class motor
```

Appendix E-2. Network Program

This program calculates currents and voltages of network buses during the simulation. It is executed with the network simulation program, Appendix E.

This program was written by Professor James L. Kirtley.

```
#include <stream.h>
#include <math.h>

class line;

class bbus                // Base class for polyphase buses
{
  double va, vb, vc;      // All buses have voltages in common

  public:
   double a_voltage()     { return(va);}
   double b_voltage()     { return(vb);}
   double c_voltage()     { return(vc);}
   void set_va(double v) {va = v;}
   void set_vb(double v) {vb = v;}
   void set_vc(double v) {vc = v;}
   void set_varc(double v) {return(va-vb);}
};

class bus : public bbus {       // network bus for time-step simulations
  line** lptr;                 // bus is connected to a bunch of lines
  int nodeno;                  // we gotta know which node we are
  int nlines;                 // and this is the number of lines
  double ia, ib, ic;           // unbalanced bus currents
  double gaai, gabi, gaci;      // inverse admittances
  double gbai, gbbi, gbci;
  double gcai, gcbi, gcci;
  double gaat, gabt, gact;
  double gbat, gbbt, gbct;
  double gcat, gcbt, gcct;

  public:
   void setbus(int node_number, int line_count, line* linebuf[]);
   void setup ();              // this step sets up node admittances
   double estimate_voltage(double dt);   // to get voltage of an isolated node
   void report();
```

```
 }; // end of declaration of class bus

class vbus : public bbus {      // voltage-source bus
  double vamp, vphase;

  public:
   void setvbus(double v, double p)
     {
          vamp = v;
          vphase = p;
     }
   void report();
   void set_v(double omt)
     {
          double s=2.0943951;
          set_va(vamp * cos (omt + vphase));
          set_vb(vamp * cos (omt + vphase - s));
          set_vc(vamp * cos (omt + vphase + s));
     }
 };                            // end of declaration of vbus

class line {
  double xs, xm, iao, ibo, ico, ia, ib, ic,
   omz, k1a, k2a, k3a, k4a, k1b, k2b, k3b, k4b,
   k1c, k2c, k3c, k4c, iap, ibp, icp;
  int nodea, nodeb, sa, sb, sc;
  bbus* busa;
  bbus* busb;

 public:
  double     gaa, gab, gac, gba, gbb, gbc, gca, gcb, gcc, r;
  void setline (int na, int nb, double x, double xz, double ra,
                    double omza, int saa, int sba, int sca,
                    double iaa, double iba, double ica);
  void set_bus_pointers(bbus* bus_a, bbus* bus_b);
  void init_currents (double iaa, double iba, double ica);
  void setup (); // Conductance Parameters
  void open_a();
  void open_b();
  void open_c();
  void close_a();
  void close_b();
  void close_c();
  void report();
  void report(char *label);
```

```
 int current_monitor ();  // Checks for zero crossings
 void re_setup (int ip);   // re-build line model
 void rk1 (double dt);
 void rk2 (double dt);
 void rk3 (double dt);
 void rk4 (double dt);
 void rk ();  // Finishes the Runge-Kutta;

 double i_a () { return (ia);}
 double i_b () { return (ib);}
 double i_c () { return (ic);}

 double ias(int nn);            // current with proper sign convention
 double ibs(int nn);
 double ics(int nn);

 double i_ap() {return(iap);}  // Partial deltas for runge-kutta step
 double i_bp() {return(ibp);}
 double i_cp() {return(icp);}

 double g_aa() {return (gaa);}
 double g_ab() {return (gab);}
 double g_ac() {return (gac);}
 double g_ba() {return (gba);}
 double g_bb() {return (gbb);}
 double g_bc() {return (gbc);}
 double g_ca() {return (gca);}
 double g_cb() {return (gcb);}
 double g_cc() {return (gcc);}

 int report_node_a();
 int report_node_b();

 double va_other_end (int thisnode);
 double vb_other_end (int thisnode);
 double vc_other_end (int thisnode);

 int report_other_node (int thisnode);

 int abs(int x) { if(x<0) return(-x); else return(x);}

};  // end of definition of class line


  void bus::setbus(int node_number, int line_count, line* linebuf[])
```

```
{
    nodeno = node_number;
    nlines = line_count;
    lptr = new line*[line_count];
    for (int j=0; j<nlines; j++)
     lptr[j] = linebuf[j];
    ia=ib=ic=0;
}

void bus::setup ()              // this step sets up node admittances
{
    gaat=0; gabt=0; gact=0;
    gbat=0; gbbt=0; gbct=0;
    gcat=0; gcbt=0; gcct=0;

    for (int i=0; i<nlines; i++)
     {
       gaat += lptr[i]->gaa;
       gabt += lptr[i]->gab;
       gact += lptr[i]->gac;
       gbat += lptr[i]->gba;
       gbbt += lptr[i]->gbb;
       gbct += lptr[i]->gbc;
       gcat += lptr[i]->gca;
       gcbt += lptr[i]->gcb;
       gcct += lptr[i]->gcc;
     }

  // invert that: since it is a 3x3, we do directly

  double det = gaat*gbbt*gcct + gabt*gbct*gcat + gact*gbat*gcbt
          - gaat*gcbt*gbct - gbat*gabt*gcct - gcat*gbbt*gact;

  gaai = (gbbt*gcct - gcbt*gbct)/det;
  gabi = (gcat*gbct - gbat*gcct)/det;
  gaci = (gbat*gcbt - gcat*gbbt)/det;
  gbai = (gcbt*gact - gabt*gcct)/det;
  gbbi = (gaat*gcct - gcat*gact)/det;
  gbci = (gcat*gabt - gaat*gcbt)/det;
  gcai = (gabt*gbct - gbbt*gact)/det;
  gcbi = (gbat*gact - gaat*gbct)/det;
  gcci = (gaat*gbbt - gbat*gabt)/det;

  cout << "bus::setup()  Here is Total Admittance G_t\n";
  cout << gaat << " " << gabt << " " << gact << "\n";
```

```
    cout << gabt << " " << gbbt << " " << gbct << "\n";
    cout << gcat << " " << gcbt << " " << gcct << "\n";

    cout << "And inverse G_0:\n";
    cout << gaai << " " << gabi << " " << gaci << "\n";
    cout << gbai << " " << gbbi << " " << gbci << "\n";
    cout << gcai << " " << gcbi << " " << gcci << "\n";
   }

  double bus::estimate_voltage(double dt) // voltage of an isolated node
   {
    double gva;
    double gvb;
    double gvc;
    int i;
    double ia, ib, ic;
    double iau, ibu, icu;     // estimated unbalance currents
    double ova, ovb, ovc;
    double va, vb, vc;          // line active voltage
    double vca, vcb, vcc;        // correction voltages
    double err;

    iau=ibu=icu=gva=gvb=gvc=0;

    ova = a_voltage();
    ovb = b_voltage();
    ovc = c_voltage();
//     cout << "bus::estimate_voltage("<< nodeno<< ")\n";

    for (i=0; i<nlines; i++)
        {
         ia = lptr[i]->ias(nodeno);
         ib = lptr[i]->ibs(nodeno);
         ic = lptr[i]->ics(nodeno);

         va = lptr[i]->va_other_end(nodeno) + lptr[i]->r * ia;
         vb = lptr[i]->vb_other_end(nodeno) + lptr[i]->r * ib;
         vc = lptr[i]->vc_other_end(nodeno) + lptr[i]->r * ic;

//           cout << "Node " << lptr[i]->report_other_node(nodeno)
//            << " V = " << lptr[i]->va_other_end(nodeno) << " "
//             << lptr[i]->vb_other_end(nodeno) << " "
//                << lptr[i]->vc_other_end(nodeno) << "\n";

          gva += lptr[i]->gaa * va
```

```
            + lptr[i]->gab * vb
            + lptr[i]->gac * vc;

           gvb += lptr[i]->gba * va
            + lptr[i]->gbb * vb
            + lptr[i]->gbc * vc;

           gvc += lptr[i]->gca * va
            + lptr[i]->gcb * vb
            + lptr[i]->gcc * vc;

          iau += ia;
          ibu += ib;
          icu += ic;
         }

      double vcac, vcbc, vccc;
      double vaf, vbf, vcf;

      vca = gaai*gva + gabi*gvb + gaci*gvc;
      vcb = gbai*gva + gbbi*gvb + gbci*gvc;
      vcc = gcai*gva + gcbi*gvb + gcci*gvc;

      vcac = - (.5/dt)*(gaai*iau + gabi*ibu + gaci*icu);
      vcbc = - (.5/dt)*(gbai*iau + gbbi*ibu + gbci*icu);
      vccc = - (.5/dt)*(gcai*iau + gcbi*ibu + gcci*icu);

      vaf = vca + vcac;
      vbf = vcb + vcbc;
      vcf = vcc + vccc;

      err = (vaf-ova)*(vaf-ova)+(vbf-ovb)*(vbf-ovb)+(vcf-ovc)*(vcf-ovc);

      set_va(vaf);
      set_vb(vbf);
      set_vc(vcf);

//      cout << "Currents " << iau << " " << ibu << " " << icu << "\n";
//      cout << "Voltages " << vca << " " << vcb << " " << vcc << "\n";
//      cout << "I Corr V " << vcac << " " << vcbc << " " << vccc << "\n";
//      cout << "Error = " << err << "\n";
      return(err);
    }

  void bus::report()
```

```
  {
   cout << "Bus " << nodeno << " Has " << nlines << " Lines\n";
   for (int j=0; j<nlines; j++)
         cout << "Line " << j <<
           " Between Nodes " << lptr[j]->report_node_a() << " And " <<
           lptr[j]->report_node_b() << " Line Pointer " << &lptr[j] << "\n";
  }

void vbus::report()
  {
   cout << "Voltage Bus V= " << vamp << " Phase= " << vphase << "\n";
  }

void line::setline (int na, int nb, double x, double xz, double ra,
            double omza=377,
            int saa = 1, int sba = 1, int sca = 1,
            double iaa = 0, double iba = 0, double ica = 0)
 {
  nodea = na;
  nodeb = nb;
  xs = (2.0 * x + xz)/3.0;
  xm = (xz - x)/3.0;
  omz = omza;
  r = ra;
  sa = saa;
  sb = sba;
  sc = sca;
  iao = ia = iaa;
  ibo = ib = iba;
  ico = ic = ica;
  cout << "Line::setline xs = " << xs << " xm = " << xm << " r = " << r
      << "\n Sw = " << sa << sb << sc << " i = " << ia << " " << ib
       << " " << ic << "\n\n";
 }

void line::set_bus_pointers(bbus* bus_a, bbus* bus_b)
  {
   cout << "line::set_bus_pointers " << bus_a << " " << bus_b << "\n";
   busa = bus_a;
   busb = bus_b;
  }

void line::init_currents (double iaa, double iba, double ica)
{
 iao = ia = iaa;
```

```
  ibo = ib = iba;
  ico = ic = ica;
  cout << "Line Init Currents " << ia << "   " << ib << "   " << ic << "\n";
}

void line::setup ()  // Conductance Parameters
{
  double d, gs, gm;
//  int abs(int);
    if (abs(sa) + abs(sb) + abs(sc) == 3)   // all in
      {
         d = xs * xs - 2.0 * xm * xm + xs * xm;
         gs = omz * (xs + xm) /d;
         gm = - omz * xm / d;
         gaa = gbb = gcc = gs;
         gab = gba = gac = gca = gcb = gbc = gm;
      }
    else if (abs (sa) + abs (sb) + abs (sc) == 2) // one line out
      {
         d = xs * xs - xm * xm;
         gs = omz * xs / d;
         gm = -omz * xm / d;
         if (sa == 0)                          // line A out
           {
             gaa = gab = gac = gba = gca = 0;
             gbb = gcc = gs;
             gbc = gcb = gm;
           }

         else if (sb == 0)                     // line B out
           {
             gbb = gab = gba = gbc = gcb = 0;
             gaa = gcc = gs;
             gac = gca = gm;
         }

         else if (sc == 0)
           {
             gcc = gca = gac = gbc = gcb = 0;
             gaa = gbb = gs;
             gab = gba = gm;
           }
         else { cout << "Blew One Line Out Case!\\n";}
      }
     else if((abs(sa) + abs(sb) + abs(sc)) == 1)
```

```
    {
       if (abs(sa) == 1)
        {
         gaa = omz/xs;
         gab = gba = gbb = gac = gca = gcb = gbc = gcc = 0;
        }
       else if (abs(sb) == 1)
        {
         gbb = omz/xs;
         gaa = gab = gba = gbc = gcb = gca = gac = gcc = 0;
        }
       else if (abs(sc) == 1)
        {
         gcc = omz/xs;
         gaa = gab = gba = gbb = gac = gca = gbc = gcb = 0;
        }
       else
        {
         cout << "Blew One Line IN case! \n";
        }
    }
   else if((abs(sa) + abs(sb) + abs(sc)) == 0)
    {
       gaa = gab = gba = gca = gac = gbb = gbc = gcb = gcc = 0;
    }
   else {cout << "Bad Combination of Switch States!\n";}

 cout << "Line::Setup() G = " << gaa << "  " << gab << "  " << gac
   << "\n                " << gba << "  " << gbb << "  " << gbc
   << "\n                " << gca << "  " << gcb << "  " << gcc << "\n\n";
}

void line::open_a() {sa = -1;}
void line::open_b() {sb = -1;}
void line::open_c() {sc = -1;}

void line::close_a() {sa = 1;}
void line::close_b() {sb = 1;}
void line::close_c() {sc = 1;}

int line::current_monitor ()  // Checks for zero crossings
{
 int ip = 0;
 if ((sa == -1) && (iao !=0) && (iao * ia <0)) ip += 1;
 if ((sb == -1) && (ibo !=0) && (ibo * ib <0)) ip += 2;
```

```
 if ((sc == -1) && (ico !=0) && (ico * ic <0)) ip += 4;

 //re-set old currents

 iao = ia;
 ibo = ib;
 ico = ic;

 return (ip);
}

void line::re_setup (int ip)   // re-build line model
{
 if (ip & 1) { sa = 0; ia = 0;} // Phase A opening
 if (ip & 2) { sb = 0; ib = 0;} // Phase B opening
 if (ip & 4) { sc = 0; ic = 0;} // Phase C opening
 setup();
}

void line::rk1 (double dt)
{
 double va, vb, vc;
 va = busa->a_voltage() - busb->a_voltage();
 vb = busa->b_voltage() - busb->b_voltage();
 vc = busa->c_voltage() - busb->c_voltage();

 k1a = gaa * (va - r * ia) * dt
    + gab * (vb - r * ib) * dt
    + gac * (vc - r * ic) * dt;

 k1b = gba * (va - r * ia) * dt
    + gbb * (vb - r * ib) * dt
    + gbc * (vc - r * ic) * dt;

 k1c = gca * (va - r * ia) * dt
    + gcb * (vb - r * ib) * dt
    + gcc * (vc - r * ic) * dt;

 iap = ia + .5*k1a;  //usable for next rk step, both internal and
 ibp = ib + .5*k1b;  //outside, reported by i_xp()
 icp = ic + .5*k1c;
}

void line::rk2 (double dt)
{
```

```
 double va, vb, vc;
 va = busa->a_voltage() - busb->a_voltage();
 vb = busa->b_voltage() - busb->b_voltage();
 vc = busa->c_voltage() - busb->c_voltage();

 k2a = gaa * (va - r * iap) * dt
    + gab * (vb - r * ibp) * dt
    + gac * (vc - r * icp) * dt;

 k2b = gba * (va - r * iap) * dt
    + gbb * (vb - r * ibp) * dt
    + gbc * (vc - r * icp) * dt;

 k2c = gca * (va - r * iap) * dt
    + gcb * (vb - r * ibp) * dt
    + gcc * (vc - r * icp) * dt;

 iap = ia + .5*k2a;  //usable for next rk step, both internal and
 ibp = ib + .5*k2b;  //outside, reported by i_xp()
 icp = ic + .5*k2c;
}

void line::rk3 (double dt)
{
 double va, vb, vc;
 va = busa->a_voltage() - busb->a_voltage();
 vb = busa->b_voltage() - busb->b_voltage();
 vc = busa->c_voltage() - busb->c_voltage();

 k3a = gaa * (va - r * iap) * dt
    + gab * (vb - r * ibp) * dt
    + gac * (vc - r * icp) * dt;

 k3b = gba * (va - r * iap) * dt
    + gbb * (vb - r * ibp) * dt
    + gbc * (vc - r * icp) * dt;

 k3c = gca * (va - r * iap) * dt
    + gcb * (vb - r * ibp) * dt
    + gcc * (vc - r * icp) * dt;

 iap = ia + k3a;  //usable for next rk step, both internal and
 ibp = ib + k3b;  //outside, reported by i_xp()
 icp = ic + k3c;
}
```

```
void line::rk4 (double dt)
{
 double va, vb, vc;
 va = busa->a_voltage() - busb->a_voltage();
 vb = busa->b_voltage() - busb->b_voltage();
 vc = busa->c_voltage() - busb->c_voltage();

 k4a = gaa * (va - r * iap) * dt
    + gab * (vb - r * ibp) * dt
    + gac * (vc - r * icp) * dt;

 k4b = gba * (va - r * iap) * dt
    + gbb * (vb - r * ibp) * dt
    + gbc * (vc - r * icp) * dt;

 k4c = gca * (va - r * iap) * dt
    + gcb * (vb - r * ibp) * dt
    + gcc * (vc - r * icp) * dt;
}

void line::rk ()  // Finishes the Runge-Kutta
{
ia += (k1a + 2.0 * k2a + 2.0 * k3a + k4a) / 6.0;
ib += (k1b + 2.0 * k2b + 2.0 * k3b + k4b) / 6.0;
ic += (k1c + 2.0 * k2c + 2.0 * k3c + k4c) / 6.0;
}

void line::report()
{
 cout << "Line From " << nodea << " To " << nodeb <<
   " r= " << r << " xs = " << xs << " xm = " << xm << "\n";
 cout << "Bus Pointers are " << busa << " " << busb << "\n";
}

void line::report(char* label)
{
 cout << "line::report " << label << "\n";
 cout << "Line From " << nodea << " To " << nodeb <<
   " r= " << r << " xs = " << xs << " xm = " << xm << "\n";
 cout << "Bus Pointers are " << busa << " " << busb << "\n";
}

int line::report_node_a() {return(nodea);}
int line::report_node_b() {return(nodeb);}
```

```
double line::va_other_end (int thisnode)
  {
//    cout << "line::va_other_end ( "<< thisnode << ")\n";
//    cout << "nodea = " << nodea << " nodeb = " << nodeb << "\n";
//    cout << "busa pointer = " << busa << " busb pointer " << busb << "\n";
//    cout << "busa v = " << busa->a_voltage() <<
//      " busb v = " << busb->a_voltage() << "\n";
    if (thisnode == nodea)
      return(busb->a_voltage());
    else return(busa->a_voltage());
  }

double line::vb_other_end (int thisnode)
  {
    if (thisnode == nodea)
      return(busb->b_voltage());
    else return(busa->b_voltage());
  }

double line::vc_other_end (int thisnode)
  {
    if (thisnode == nodea)
      return(busb->c_voltage());
    else return(busa->c_voltage());
  }

int line::report_other_node (int thisnode)
{
  if (thisnode == nodea)
    return(nodeb);
  else return(nodea);
}

double line::ias(int thisnode)
  {
    if (thisnode == nodea)
      return(ia);
    else return(-ia);
  }

double line::ibs(int thisnode)
  {
    if (thisnode == nodea)
      return(ib);
```

```
  else return(-ib);
 }

double line::ics(int thisnode)
 {
  if (thisnode == nodea)
    return(ic);
  else return(-ic);
 }
```

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOl
MO[illegible] [illegible]-5101

GAYLORD S

DUDLEY KNOX LIBRARY
3 2768 00310833 3